2

A RAND NOTE

The RAND-ABEL® Programming Language: Reference Manual

Norman Z. Shapiro, H. Edward Hall,
Robert H. Anderson, Mark LaCasse,
Marrietta S. Gillogly, Robert Weissler

December 1988

40 Years 1948-1988

RAND

The research described in this report was sponsored by the Director of Net Assessment, Office of the Secretary of Defense (OSD), under RAND's National Defense Research Institute, an OSD-supported Federally Funded Research and Development Center, Contract No. MDA903-85-C-0030.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

A RAND NOTE

N-2367-1-NA

The RAND–ABEL® Programming Language: Reference Manual

Norman Z. Shapiro, H. Edward Hall,
Robert H. Anderson, Mark LaCasse,
Marrietta S. Gillogly, Robert Weissler

December 1988

Prepared for
The Director of Net Assessment,
Office of the Secretary of Defense

A research publication from
The RAND Strategy Assessment Center

40 Years 1948-1988

RAND

PREFACE

The RAND-ABEL®[1] was developed at The RAND Corporation, originally for use in writing complex decision-model "agents" as part of a knowledge-based simulation for automated war gaming. It was designed and implemented initially by Norman Z. Shapiro, H. Edward Hall, and Mark LaCasse. Robert Weissler has made important subsequent contributions.

RAND-ABEL is an evolving *operational* language that is now being used in a number of diverse projects. It will be available in the public domain. This Note, which updates a 1985 publication, documents the RAND-ABEL language as it existed in March 1988. It is intended primarily for programmers. It gives a terse but complete description of the language. It assumes the reader is fluent in at least one high-level programming language and is familiar with the notation and concepts used to describe the formal syntax of programming languages. For background on RAND-ABEL's origin and underlying principles, see:

Shapiro, Norman Z., H. Edward Hall, Robert H. Anderson, and Mark LaCasse, *The RAND-ABEL™ Programming Language: History, Rationale, and Design*, The RAND Corporation, R-3274-NA, August 1985.

Current plans are to complete by the end of 1988 the addition of several new RAND-ABEL features. These new features will broaden the scope of problems addressed by RAND-ABEL, and allow for clearer and more efficient modeling. The features include: sets (with enumerative values as legal members), type-unions (with dynamic type assignment and checking), data structures of arbitrary complexity, and lists (including a variety of operators for manipulating them). Readers having versions of RAND-ABEL produced after this manual's publication should check the relevant on-line documentation to see what additional features their software supports (see file UPDATES in the main RAND-ABEL source directory).

[1]RAND-ABEL is a trademark of The RAND Corporation.

This work was conducted by the RAND Strategy Assessment Center and sponsored by the Director of Net Assessment in the Office of the Secretary of Defense, under the auspices of RAND's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense.

Inquiries and comments are welcome. They may be sent directly to the authors or to Dr. Paul K. Davis, Director of the RAND Strategy Assessment Center.

SUMMARY

This reference manual describes the RAND-ABEL programming language. In designing the RAND-ABEL language, we determined that

- RAND-ABEL should be suitable for large, rule-based systems.
    - It should lend itself to program development by multimember programming teams.
    - It should be relatively easy to maintain.
- RAND-ABEL should be suitable for war gaming and multiscenario sensitivity analysis.
    - Domain-substantive RAND-ABEL rules should be readable by domain specialists who are not RAND-ABEL programmers, and the code should be relatively self-documenting.
    - RAND-ABEL should be efficient in execution.
- RAND-ABEL should be suitable for use by any of several governmental gaming and analysis organizations.
    - It should be transportable to various computers capable of hosting the UNIX operating system.

The RAND-ABEL language was designed for the specific requirements of the RAND Strategy Assessment Center (RSAC). The RSAC has developed a large system for automated and semiautomated war gaming in which separate models represent U.S., Soviet, and third-country behavior.[1] RAND-ABEL is a preprocessor for the C programming language under the UNIX operating system, which makes RAND-ABEL quite portable across different computers. RAND-ABEL is very fast in execution time compared with other languages of similar readability. We estimate that C language programs execute no more than three times faster than comparable RAND-ABEL programs.

[1]See Davis and Winnefeld, 1983; Davis, Bennett, and Schwabe, 1988; Davis, 1988; and Davis and Hall, forthcoming.

In the RAND Strategy Assessment System (RSAS) environment, RAND-ABEL is used with a data dictionary, a data editor, and support for coroutines. This allows a flexible, hierarchical modeling system, allowing human teams to replace some of the models. Although designed for the RSAS, we anticipate that RAND-ABEL will be of interest for other applications on UNIX systems requiring a highly readable language, fast performance, and early discovery of errors (for example, in the design of large rule-based models and simulations).

The RAND-ABEL language provides a number of unique capabilities, including support for tables within the source code. Tables can be used as decision tables or to govern an iterative execution. We find that table statements provide a much more succinct and readable alternative to long sequences of sentence-like rules typical of rule-based languages. When used as decision tables, RAND-ABEL tables correspond closely to the decision trees analysts and reviewers use in working out a logically complete argument. RAND-ABEL tables have a syntax that is inherently two dimensional.

RAND-ABEL is a strongly typed language, permitting certain types of errors in complex programs to be uncovered early. Many of RAND-ABEL's features are derived from constructs in the C programming language.

ACKNOWLEDGMENTS

Joan LaCasse tested the RAND-ABEL Translator's robustness and diagnostics at various stages, helped debug some of the code, and wrote preliminary documentation.

Several other members of the RAND Strategy Assessment Center (RSAC) staff have made valuable suggestions regarding RAND-ABEL syntax and requisite features. These include Steven Bankes, Arthur Bullock, Paul Davis, William Jones, Christe McMenomy, Ross Quinlan, and Herbert Shukiar.

We especially wish to acknowledge strong and continuing support for the development of the RAND-ABEL language by Paul Davis and Herbert Shukiar of RSAC. Paul's stubborn refusal to be satisfied with anything less than our best effort at meeting RSAC's real needs, Herb's effective and knowledgeable guidance, and their allocation of resources to RAND-ABEL when its future was uncharted are responsible for RAND-ABEL's success to date.

CONTENTS

I. INTRODUCTION

RAND-ABEL is a computer programming language implemented on the UNIX[1] operating system. A program called the "RAND-ABEL Translator" compiles RAND-ABEL statements into a C program (Kernighan and Richie, 1978), which is in turn compiled and run.

RAND-ABEL was developed at the RAND Strategy Assessment Center (RSAC), to be used in the development of complex models. Six primary design goals guided the development of RAND-ABEL. RAND-ABEL is intended to be:

- Reasonably self-documenting. The RAND-ABEL code, by itself, should convey the meaning of a program.
- Understandable by English speakers familiar with the subject matter. Readers of the program should not need detailed programming knowledge to comprehend the program.
- Reasonably easy to learn and use by individuals with good analytic capability and modest programming skills. Analysts and application specialists with only some prior experience in a high-level programming language, such as FORTRAN, should be able to program effectively in RAND-ABEL without extensive training and study.
- Rapid in execution. Since RAND-ABEL was specially designed for building rule-based programs with many qualitative variables, it is important that these large programs be able to execute rapidly and efficiently.
- Portable across different types of computer hardware. RAND-ABEL and systems developed in it should not be unique to a single computer or manufacturer's computers but rather be portable across a range of minicomputers and powerful microcomputers.
- Supportive of specialized needs of the RAND Strategy Assessment System (RSAS), such as coroutines and tabular data, and well-suited to the creation of complex simulations by groups of developers.

[1]UNIX is a trademark of AT&T Bell Laboratories.

Although reading and changing RAND-ABEL programs is relatively easy and within the capabilities of many analysts who know some other computer language, some RAND-ABEL programming, such as changing the RSAS Data Dictionary, requires relatively high levels of programming skill and knowledge. RAND-ABEL is not unique in requiring high skill levels in order to exploit the full capabilities of the language; however, its friendly readability can give an erroneous impression regarding writability.

Prior to the development of RAND-ABEL, the RAND Strategy Assessment Center used the ROSIE[2] language (Fain et al., 1981) for programming the Scenario Agent. The ROSIE program representing the Scenario Agent of the Mark II RSAC system is documented in Schwabe and Jamison (1982). We found that ROSIE was too slow for our future needs, which included operating large-scale simulations in a matter of minutes. Also, we sought special features such as decision tables that were not likely to be available in ROSIE. Therefore, RAND-ABEL was developed as a separate language.[3] Nevertheless, much of the form and style of the statements in RAND-ABEL derives from its ROSIE heritage. Because analysts were using only a portion of the features of ROSIE, it was possible to design RAND-ABEL as a simpler language.

The goals of speed in execution and portability were achieved by writing a RAND-ABEL compiler (called the RAND-ABEL Translator) that translates RAND-ABEL statements into statements in the C language. Because C and its host operating system, UNIX, are available on many different computers, RAND-ABEL is similarly portable. In addition, efficient C language compilers are available, thereby permitting the efficient compilation of RAND-ABEL statements through this two-step compilation process. As a result, RAND-ABEL can be used on any computer running the UNIX 4.2bsd operating system and having a C language compiler. Moving RAND-ABEL to later versions of AT&T's UNIX system should be trivial.

---

[2]ROSIE is a registered trademark of The RAND Corporation.

[3]The history, rationale, and design of RAND-ABEL is described in Shapiro et al., 1985.

RAND-ABEL has proven to be exceptionally useful for knowledge-based modeling and has been used to generate approximately 250,000 lines of code representing models dealing with political-military decisionmaking, situation assessment, command-control, and adjudication of results of force interactions. It should be noted, however, that RAND-ABEL is a procedural language and does not have an "inference engine." It is therefore quite suitable for representing knowledge of the form If <situation> Then <action>, but it does not have the particular inferencing capabilities of, for example, LISP, PROLOG, and ROSIE.

RAND-ABEL is a strongly typed language; that is, the properties of all identifiers are declared before they are used. The RAND-ABEL compiler uses these properties to test the validity of RAND-ABEL statements so that certain types of errors (particularly control-related and data-related errors) may be caught at the earliest possible time.

The RAND-ABEL language contains a number of novel features. Perhaps the most novel construction in RAND-ABEL is the table statement, described in detail in Sec. VII of this manual. The following is valid RAND-ABEL code. (In this and other examples within this manual, RAND-ABEL keywords are printed in boldface to distinguish them from identifiers, constants, and comments chosen by the user to describe a particular application. Comments appear within square brackets.)

```
Table Deploy
  [This table orders deployment of forces]

   qty    #-%  unit-type  unit-owner      to-area
  =====   ===  =========  ==============  ========= .
     20    %   Troops     Denmark         CEur-res
     20    %   Troops     Netherlands     CEur-res
     20    %   Troops     FRG             CEur-res
      1    #   Troops     UK              CEur-res  .
```

All RAND-ABEL tables consist of column headings followed by rows containing data. In this example, there are five columns. The meaning of the top row of the table, as defined by the Deploy function, orders 20 percent of the troops "owned by" Denmark to deploy to CEur-res

(Central European theater reserves). The table calls the Deploy function four times (once for each row of data) with five parameters (corresponding to the entry in each column of the table). On the fourth call, one division of British troops (the "#" means number, rather than percent) are deployed.

The table statement is a powerful device, capable of both defining iterative processes and creating decision tables. Its syntax is fully two dimensional. A function call or RAND-ABEL statement (possibly a compound statement) occurring immediately after the table keyword is called once for each row in the table, with the table's column headings being parsed and matched with function parameters or variables in the statement. The table statement was developed because tables of information are commonly used by many types of planners and analysts.

A second noteworthy feature of RAND-ABEL is "declaration by example." All identifiers are declared by giving examples of their use, usually by an assignment statement such as:

Declare message by example: Let message be "I have Checkmate".

or more briefly

Declare message: Let message be "I have Checkmate".

which declares the variable "message" to be a character string. In this manner, the data type associated with an identifier is declared without requiring the use of a whole vocabulary (e.g., integer, real, character string, Boolean, process, enumerated variable, array) that may not be meaningful to analysts who are not professional programmers. Furthermore, it is especially useful in rule-based systems with many ad hoc data types that otherwise would require names for strong typing (i.e., the data types to which enumerated variables belong).

RAND-ABEL also has a built-in set of functions to handle coroutines and a "data dictionary" to coordinate external data references among program modules being developed independently. These language features are discussed in Secs. IX and X of this manual.

NOTATIONAL CONVENTIONS

This manual presents the form and content of the RAND-ABEL programming language. In doing so, it must use a set of stylistic conventions to represent RAND-ABEL's form. These conventions must not be confused with the form of the RAND-ABEL syntax itself. The conventions used in this document are:

1. The RAND-ABEL language relies on a number of special keywords, or reserved words, which have a particular meaning. Appendix B contains a complete list of RAND-ABEL keywords. As mentioned earlier, in this manual RAND-ABEL keywords are printed in boldface, to distinguish them from other language constructs. In a RAND-ABEL program, these keywords must be written in lower-case, with the first letter optionally upper-case. Therefore, the only two valid ways of writing a keyword are:

   Declare, declare

   This same case freedom does not extend to RAND-ABEL identifiers or character strings. The variable name "Country" is distinct from the (dangerously similar) name "country".

2. The syntax of RAND-ABEL is sometimes best described in terms of a set of syntactic categories, which themselves have a defined structure. These syntactic categories are represented by a word or phrase in italics, like *expression*. What is allowed in place of these categories is described in various sections of this manual. To find the definition of one of these categories, look under that category in the index to this manual; it tells on which page that definition occurs. A list of syntactic categories is given in Appendix B.

3. To represent a set of options, one of which must be chosen, we use a single-spaced vertical stack of options. For example,

   Trace If.
   Trace Function.

   This notation means that the TRACE statement can take the forms "Trace If." or "Trace Function.". To represent a continuation of the previous line, we indent the second part of the definition. For example,

   Declare *variable-name:*

   Let *variable-name* be *expression*.

Let *variable-name* be *identifier constant*.
Let *variable-name* be *enumerated variable*.

indicates that the phrase "Let *variable-name* be" is a required part of the Declare statement, and that it must be followed by a representative of one of the three syntactic categories listed: *expression*, *identifier constant*, or *enumerated variable*. One valid form of the Declare statement is therefore

Declare *variable-name:* Let *variable-name* be *expression*.

4. Ellipses (i.e., "three dots" notation) are used to represent a sequence of zero or more RAND-ABEL constructs. For example,

   statement . . . *statement*

   means that zero or more *statements* can occur in a sequence. By extension, if a delimiter is used after the first occurrence and before the second occurrence with the three-dot notation in between, it means zero or more instances of that construct can occur in sequence, separated from each other by that delimiter. For example,

   name , . . . , *name*

   means that zero or more RAND-ABEL *names* can occur, separated by commas (the delimiter here). When ONE or more occurrences are required, the above notation is sometimes used for convenience, with a note immediately below the syntax diagram stating that restriction.

5. The structure representing a particular RAND-ABEL language category is boxed so that it can be found easily. Rules specifying various restrictions and notes regarding this structure then follow. The particular syntactic category being (perhaps partially) defined is shown within the top border of the box. For example:

```
+-statement---------------------------------------------------+
|                                                             |
|     If Boolean-expression Then statement                    |
|                                                             |
|     If Boolean-expression Then statement Else statement     |
|                                                             |
+-------------------------------------------------------------+
```

II. NAMES, IDENTIFIERS, WHITE SPACE, AND COMMENTS

NAMES AND IDENTIFIERS

The entities of a RAND-ABEL program (e.g./variables) have names. Each is represented by a RAND-ABEL *identifier*, which is composed of a sequence of one or more of the following characters, without intervening spaces:

upper- and lower-case letters (A-Z, a-z)
digits (0-9)
the hyphen (-) (or its synonym, the underscore (_))
the number or pound sign (#)
the percent sign (%)
the plus sign (+)
the ampersand sign (&)
the slash (/)
the period (.)

Rules:

1. An identifier cannot end with a period.

2. An identifier should not begin with a hyphen (-), its synonym underscore (_) or a period (.).

3. An identifier cannot extend over a line of text (i.e., it cannot contain a carriage-return or line-feed character). It also cannot extend over a line of text by being hyphenated; the hyphen is treated just as any other character. (An identifier used as a column heading in a table statement is an exception to this rule. See the discussion of the table statement in Sec. VII.)

4. Upper-case letters are distinct from lower-case letters within identifiers; for example, the following identifiers represent different data items:

   Country, country, COUNTRY, CounTry

   Within RAND we recommend a programming style in which global variables are capitalized and local variables are not. For instance, "British-mood" indicates a global variable, while "british-mood" signifies a local variable. It should be noted that this is a programming convention, not a requirement of RAND-ABEL syntax.

5. A sequence of characters meeting the above restrictions, and intended as an identifier, must also not be recognizable as anything else, such as an integer or real number.

There is no restriction on length, other than the one-line limitation (Rule 3 above).

It should be noted that various text formatters in use at RAND and elsewhere interpret a period (.) in column 1 as a special formatting instruction, thereby causing problems if that is not intended. No RAND-ABEL statement begins with a period, but one could inadvertently appear in column 1 if an identifier begins with a period and a statement is continued onto a following line, causing an identifier to appear first on the succeeding line. It is safest not to start any identifier with a period.

The C compiler generates variables that begin with initial hyphens (-) and initial underscores (_). To preclude the possibility of confusion, it is recommended that RAND-ABEL identifiers avoid this usage: i.e., do not begin an identifier with a hyphen or an underscore.

Examples of valid RAND-ABEL identifiers:

```
country
Order-WTVD1-force-assignment
assumptions-re-Europe-On-Call
84flight#-2
```

An identifier, having the form described above, can also be an "identifier constant" as a member of the range of an enumerated data type. See Sec. III for more information on enumerated data types.

In RAND-ABEL variables and functions may be assigned attributes such as ownership. Variables may be global, owned, or local; functions may be owned or global. The majority of variables and functions in the RSAS are owned. Ownership is described in detail in Sec. IX.

A NOTE ON WHITE SPACE AND COMMENTS

RAND-ABEL statements and definitions consist of a set of words, some of them reserved keywords, some of them names (of variables, functions, etc.), and some of them unexecuted noise words or comments, made up by the program's author. The following rules hold in writing RAND-ABEL programs. Since RAND-ABEL is built upon the C language, the C conventions for program form should be followed in case of uncertainty.

Form Rules:

1. One or more spaces separating RAND-ABEL language constructs is considered to be "white space" that acts as a separator. Any comment enclosed by square brackets is also considered to be white space. Examples:

```
Let [the variable] Country be US.
Let[the variable]Country be US.
```

   Both of these statements are equivalent to the statement:

```
Let Country be US.
```

   White space occurring within a RAND-ABEL table header has special meaning; these rules do not strictly apply within it. See the subsection "Table Statement" in Sec. VII for further information.

2. Carriage returns or line feeds are equivalent to space characters in creating white space; they have no other syntactic or semantic meaning. Examples:

```
Declare [the function] plan by example:
      Let [the] plan of US,   [the originator]
                  [in] 1984,  [the time period]
              [within] Europe [the locale]
                     be Defend-borders.
```

   The above statement is equivalent to the statement:

```
Declare plan by example:
     Let plan of US, 1984, Europe be
                              Defend-borders.
```

Spaces and other characters occurring within a string enclosed in double quotes (") are taken literally and are not considered white space as the term is being used in this discussion. For further information about character strings, see Sec. III or the syntax of character strings in the C language definition. Also, spaces are treated specially within table headers. See the discussion of the table statement for further information.

III. DATA TYPES

BASIC DATA TYPES

The RAND-ABEL language recognizes nine basic data types. In addition, the use of enumerated data types and the ability to create pointers and arrays allow the user to construct an arbitrary number of additional data types. Every simple variable, value, and expression in the language is of one of these data types, either through explicit declaration or (in the case of expressions) by derivation from the form of constants and the prior declaration of variables used within them.

The nine basic RAND-ABEL data types are shown on page 12. Integer, real, string, Boolean, process, and stream data types are built into the RAND-ABEL language. Enumerated data types, pointers, and arrays are constructed data types. Process and stream data types are usually hard-wired into RSAS underlying code where casual users do not encounter them.

Strong Typing

RAND-ABEL is a "strongly typed" language. That is, the data variables, values, or expressions on either side of an assignment statement, or binary operator, or used in place of a function's formal parameter, must agree. This strong typing is possible because all identifiers must be declared explicitly prior to use, thereby associating the identifier with a data type (or in the case of a structure or function, a sequence of data types). The RAND-ABEL translator will flag a statement as being in error if there is a mismatch of data types within the statement. (The only exception to this statement relates to the integer data type as explained under "Comparison Operators" in Sec. IV.)

Data Type	Description	Example(s)
1. Integer[1]	Whole number with + or - prefix optional (no intervening space) and no decimal point explicitly given.	1, -3567, 0, +45
2. Real[2]	Decimal numeric value with + or - prefix optional (no intervening space) and with decimal point explicitly positioned.	5.34, -.0079, 0.0, 8. 6.02 E 23 (= 6.02 x 10**23) 4 E 3 (= 4000.0) 4 E -3 (= .004)
3. String[3]	String of zero or more characters delimited by quotation marks.	"Ally is not responsive." ""
4. Boolean	A logical data item that can take on only one of the two values: Yes, No.	Yes, No
5. Enumerated	An explicit, finite-ordered list of values, consisting of RAND-ABEL identifiers.	Red, Blue, Green USA, France, FRG
6. Array	A table of values of one or more dimensions, indexed by integers and/or enumerated data type(s).	
7. Pointer	A variable whose value is the address of a variable or function.	
8. Process[4]	An identifying number for a RAND-ABEL coprocess. Two reserved RAND-ABEL keywords represent the current process (Self) and its parent (Parent).	
9. Stream	Output file pointers.	log-file

[1]Integers are implemented in RAND-ABEL as the C language data type "long int", and are therefore subject to C restrictions for that data type.

[2]These numbers must have a decimal point contained in them, or use "E" (exponential) notation to represent a power of 10. If they use "E" notation, the number following the "E" must be a whole number. Since use of the E operator could be construed as part of an identifier (e.g., in 4E3 or even 4.0E3), it must be separated from its arguments by white space as shown in the examples above. Real numbers are implemented in RAND-ABEL as the C language data type "float", (i.e., single precision floating-point numbers) and are therefore subject to C restrictions for that data type.

[3]A string may be up to 256 characters in length and may contain embedded special characters such as carriage returns and line feeds. These are specified by the special escape sequence backslash (\), such as \n for line feed (see "Format Specification" in Sec. VII). The normal syntax and rules for character strings in the C language apply for character strings in RAND-ABEL.

[4]How coprocesses are started and manipulated is described in Sec. X.

In fact, the strong typing goes considerably deeper for constructed data types. Consider the following examples:

- A pointer is declared to point to an array of real numbers. The array has two indices: an integer and an enumerated data type. All assignments of this pointer to other data constructs must retain all of these characteristics: a two-dimensional array storing real numbers, indexed by an integer and that same enumerated data type, respectively.

- A function is declared to return a process as its value and has three formal parameters: a Boolean, a character string, and a pointer to an array having the characteristics given in the previous example. All calls on this function must meet all these data type constraints, including the correct data types on the indices and stored values of the array pointed to by the third parameter.

ENUMERATED DATA TYPES

In addition to the six built-in data types, additional data types may be constructed by the following mechanisms: enumerated data types, arrays, and pointers to various data constructs.

Enumerated data types are constructed by definitions of the following form:

```
Define Enumeration Type-country: France, Germany.
```

This defines a new data type, Type-country, that can take on exactly two values: the identifiers France and Germany. These values will be called "identifier constants" within this document.[5]

[5]Definitions of enumerated data types are always made in the Data Dictionary files. (See Sec. IX.)

Enumerated types are used in declarations[6] of the form:

```
Declare country by example: Let country be Type-country.
Declare Maginot-line: Let Maginot-line be Type-country.
```

These declarations create the two distinct variables "country" and "Maginot-line" that can each take on only the values listed in the definition of Type-country: either "France" or "Germany". (See Sec. V for a description of declaration options.) The phrase by example is optional. It should be noted that a declaration is NOT an assignment statement. In the above examples, neither variable has any value yet.

Enumerated data types are simply data types that take on an explicit, finite-ordered list of values. The values are simple RAND-ABEL identifiers. In the above example, because France is listed first, it can be said to be "less than" Germany. This property can be used effectively in conditional statements when a specific portion of the list of the enumerated data type needs to be referenced.

The only way an identifier constant is established is by the declaration of an enumerated data type, including that identifier constant in its range (i.e., as part of the specified list). The set of valid identifier constants is given in a definition of the form:

Define Enumeration Type-color: Red, Green, Blue.

In this example, "Type-color" is a new data type, and "Red," "Green," and "Blue" are the (only) identifier constants in its range.

There is one reserved enumerated value that is a member of all enumerated data types, even though it is not explicitly listed. It is the value Unspecified. Any enumerated data type can take on this value. The value can be explicitly given to an enumerated type through use of the keyword Unspecified (or its synonym "--") used in place of an

[6] Declarations using enumerated data types are usually made in the Data Dictionary files, unless the variable is a local variable. For local variables, declarations are made in the relevant RAND-ABEL source code.

identifier constant in an assignment statement. Note that it is NOT automatically assigned by RAND-ABEL, however. Enumerated types have NO value before one is explicitly assigned (i.e., Unspecified is NOT the default value for enumerated types).

All enumerated data types are distinct from one another. Therefore, the value of one enumerated data type cannot be assigned to another (because this violates the "strict type checking" rule that only the same data types may be compared, assigned, or operated on together). Furthermore, all identifier constants are distinct from each other. For example consider the two declarations:

```
Define Enumeration Type-color: Red, Green, Blue.
Declare color: Let color be Type-color.

Define Enumeration Type-mood: Angry, Blue, Querulous.
Declare mood:  Let mood be Type-mood.
```

If the two assignment statements were executed:

```
Let color be Blue.
Let mood  be Blue.
```

then not only is it NOT true that the value of color equals the value of mood (because the two Blues are distinct identifier constants), BUT IN ADDITION, THEY CAN'T EVEN BE COMPARED, as in:

```
If color = mood Then ...
```

because, as stated earlier, color and mood are two distinct data types, and therefore it is illegal to compare them.

IV. VALUES, EXPRESSIONS, AND OPERATORS

VALUES AND SIMPLE EXPRESSIONS

In a programming language, a value is informally considered to be a simple term that can be evaluated to yield either a storage location or the contents of that location. If it appears on the lefthand side of an assignment statement, its evaluation yields a location at which the assignment is to be made; if it appears on the righthand side of an assignment, or elsewhere, its evaluation yields a data value. The description of the syntax of RAND-ABEL relies on the following categories of values and expressions, which are explained in this section:

Name	Informal meaning
array-access	Access to the value stored at one of the cells in an array.
lvalue	A reference that can appear on the lefthand side of an assignment statement; that is, it designates a storage location at which a value is located.
unitparam	A value that is assigned to a parameter in a function call. A simple value, or else a parenthesized expression.
simple-expr	Any of the above values, or in addition a pointer to a function.
expression	A value, or a sequence of values related by operators.

The following tables give more precise definitions for these terms, nested to show the manner in which some definitions include others. For example, since the boxes defining *unitparam* and *lvalue* are contained within the box labeled *simple-expr*, all of the varieties of *unitparam* and *lvalue* (defined by the contents of their boxes) can be used wherever a *simple-expr* is needed. Similarly, the various types of *array-access* can be used whenever an *lvalue* is needed.

```
+-expression-------------------------------------------------------+
|                                                                  |
|  Report from function-invocation                                 |
|                                                                  |
|  Evaluate unitparam  . . .  unitparam                            |
|  Evaluate with format-spec unitparam  . . .  unitparam           |
|                                                                  |
|  unary-operator expression                                       |
|  expression binary-operator expression                           |
|                                                                  |
|  simple-expr                                                     |
|                                                                  |
+------------------------------------------------------------------+
```

Rules:

1. The Report from *function-invocation* expression calls the named function and returns as its value the value returned by the function. Example:

   ```
   Let message be Report from plan of US, 1984, Europe.
   ```

2. The Evaluate expression allows the RAND-ABEL writer to generate a string of characters (usable in further RAND-ABEL processing) from a sequence of arguments, each of them a *unitparam*. This string may then be used in subsequent RAND-ABEL statements, for example, as an argument to a function that requires a string as one of its parameters. The *format-spec* is a string of characters that can control the formatting of the resultant string; the syntax and options available for a *format-spec* are described in the subsection "Input/Output" in Sec. VII. Example:

   ```
   Perform data-logging
       using Evaluate "The ally of" country "is"
                      (ally of country)
       as message.
   ```

In the above example, note that blanks are not required within the character strings to prevent the value of country from running into "The ally of" and "is"; this is because the default print format for an enumerated variable contains prefix and postfix blanks. (See "Default Output Formats" within the "INPUT/OUTPUT" subsection of Sec. VII.)

A *simple-expr* is defined by the following nested set of tables. The nesting again shows that certain terms are contained within the definitions of others. For example, a *unitparam* is one valid type of *simple-expr*, so any of the methods of constructing a *unitparam* can be used wherever a *simple-expr* is needed.

```
+-simple-expr-----------------------------------------------+
|                                                           |
| Function function-name                                    |
|                                                           |
|  +-unitparam------------------------------------------+   |
|  |                                                    |   |
|  |  Yes                                               |   |
|  |  No                                                |   |
|  |  enumerated-value                                  |   |
|  |  numeric-literal                                   |   |
|  |  quoted-string                                     |   |
|  |  variable-name                                     |   |
|  |  Unspecified                                       |   |
|  |       --                                           |   |
|  |       **                                           |   |
|  |  ( expression )                                    |   |
|  |                                                    |   |
|  +----------------------------------------------------+   |
|  +-lvalue---------------------------------------------+   |
|  |                                                    |   |
|  |   variable-name                                    |   |
|  |                                                    |   |
|  |   Value of variable-name                           |   |
|  |   Value of array-access                            |   |
|  |                                                    |   |
|  |   Pointer to variable-name                         |   |
|  |   Pointer to Attribute array-name                  |   |
|  |                                                    |   |
|  |   +-array-access---------------------------------+ |   |
|  |   |                                              | |   |
|  |   | array-name of simple-expr ,        . . .     | |   |
|  |   |            in             , and              | |   |
|  |   |            by             , and              | |   |
|  |   |                                              | |   |
|  |   |                   . . .   ,   simple-expr    | |   |
|  |   |                           , and              | |   |
|  |   |                                              | |   |
|  |   +----------------------------------------------+ |   |
|  +----------------------------------------------------+   |
+-----------------------------------------------------------+
```

Rules:

1. The Function keyword, followed by the name of a RAND-ABEL function returns as its value the address of that named function. This returned value is of type pointer. Note: Function is an alias for Pointer to.

2. "--" is a synonym for the keyword Unspecified, which is used in conjunction with enumerated data types. (See the discussion of enumerated data types in Sec. III.) In tables, the "don't care" symbol "**" is also available.

3. The *variable-name* or *array-access* following the keywords Value of[1] must be of data type pointer. The clause returns the contents of the storage location pointed at by that pointer.

4. The Pointer to[2] clause yields a value of type pointer. For example:

   Pointer to Attribute country-array

   returns a pointer to the array named "country-array".

OPERATORS

The operators used to construct expressions can be categorized as numeric operators, comparison operators (providing equality and inequality tests), logical operators (yielding a Yes or No result), and string operators. Each of these categories is described below. In each case, we list a "preferred form" for representing these operators to create as much consistency and readability in RAND-ABEL programs as possible.

[1]The Value of keyword is identical in meaning to Occupant of used in earlier versions of RAND-ABEL. Both keywords are currently supported, but Occupant of is being phased out.

[2]The Pointer to keyword has the same meaning as Address of in earlier versions. Both keywords are currently supported, but Address of is being phased out.

Numeric Operators

Mathematical Notation (Preferred Form)	"English-like" Notation	Meaning
+	plus	Addition
-	minus	Subtraction
*	times	Multiplication
/	divided by	Division
	modulo	Modulo
-	negative	unary "-" sign

When an integer variable accepts the result of the division of two integers, the result will be truncated toward zero to an integer. Examples:

73/10 evaluates to the value 7

-73/10 evaluates to the value -7

Division by zero results in a run-time error.

These operators can yield floating point exceptions (i.e., error conditions) in a machine-dependent manner.

The modulo operator returns the remainder upon division. Example:

23 modulo 8

is 7. The modulo operator requires integer arguments.

Only integer and real data types may be used as arguments for these operators. (Exception: the more stringent requirement for modulo

Comparison Operators

Comparison operators are categorized below as "equality" or "inequality" type operators. All comparison operators yield a value that is Boolean. It is permissible that one of the operands to these comparison operators have an ambiguous data type, IF that ambiguity can be resolved by the requirement of consistency with the other operand's data type. RAND-ABEL does not, however, accept two operands of ambiguous data type and attempt to resolve the mutual ambiguity. (Ambiguity can arise from a value like the identifier constant Blue, if more than one enumerated data type contains this constant in its range.)

Equality Tests

Mathematical Notation (Preferred Form)	"English-like" Notation	Meaning
=	is	Is equal to
~=	is not are not	Is not equal to

Both arguments to these operators must have the same type (with one exception: an integer appearing where a real is needed is interpreted as a real for that purpose).

Two operands from any one data type may be compared using these equality operators.

Two strings are equal only if they have the same length (including possibly zero length--i.e., the null string) and, at each respective character position, their corresponding characters are equal. (Upper-case and lower-case versions of a character are treated as different characters in this test.)

Two enumerated values are equal only if they are represented by the same identifier constant and are in the range of the same (enumerated) data type. It is an error if two enumerated values are compared that belong to different (enumerated) data types. The reserved word Unspecified is the only exception to this rule: Any enumeration may be compared with Unspecified.

Inequality Tests

Mathematical Notation (Preferred Form)	"English-like" Notation	Meaning
>=	is at least	Greater than or equal to
<=	is at most	Less than or equal to
>	is greater than	Greater than
<	is less than	Less than

Both arguments to these operators must have the same type (with one exception: an integer appearing where a real is needed is interpreted as a real for that purpose).

The following data types may be compared using the inequality operators: integer, real, string, enumerated. Note that Boolean data types *CANNOT* be compared using these operators.

Integer and real data types compare according to their values.

Strings a and b compare as follows. (Note: in comparing two individual characters, the collating sequence for the individual computer on which RAND-ABEL resides is used; this test is therefore implementation dependent.)

a. If a and b are both the null string, they are equal.

b. If one of them is the null string and the other is not, then the null string is less than the other.

c. Otherwise, compare strings a and b character by character; if each of these comparisons is **Yes** (i.e., true) at the time the end of one of the strings is reached, but the other string still has additional characters, then the shorter string is less than the longer one.

d. Otherwise, compare strings a and b character by character; if each of these comparisons is **Yes** (i.e., true) up to character position k, but is **No** (i.e., false) at character position k+1, then if string a's character in position k+1 is less than, or greater than, string b's character in position k+1, then string a is less than, or greater than (respectively) string b.

Enumerated values compare with the inequality operators according to the following rules:

a. If either or both values is **Unspecified** (or its synonym "--"), then the result is **No**.

b. If the values belong to different (enumerated) data types, it is an error.

c. Identifier constant C1 is less than identifier constant C2 if and only if C1 appears before C2 in the sequence of identifiers defining the range of their common (enumerated) data type. If they are the same identifier within this data type, they are equal. For example, for the enumerated data type defined as

   Define Enumeration Type-color: Red, Green, Blue, Purple.

   comparisons will show

```
Red < Green
Purple > Blue
Green = Green
```

Logical Operators

Logical operators are used to combine two different Boolean operands--that is, ones taking the values **Yes** or **No**--to yield a new Boolean value. For example, a RAND-ABEL program might require the logic:

If agreement and (Red-violates or Blue-violates)
Then Let agreement be No.

The assignment of the value No to the variable "agreement" will take place only if the existing value of agreement is Yes, and in addition either "Red-violates" or "Blue-violates" (or both) is Yes.

Mathematical Notation	"English-like" Notation (Preferred Form)	Meaning
&	and	Logical "and"
|	or	Logical "or"
~	not	(unary) Logical "not"

The meanings of these operators are given by the following table:

a	b	not a	a and b	a or b
yes	yes	no	yes	yes
yes	no	no	no	yes
no	yes	yes	no	yes
no	no	yes	no	no

The logical operators require Boolean values (that is, Yes or No) as their arguments and return a Boolean value as the result.

Remember, Boolean variables are not enumerations. That is, they cannot be Unspecified. In tables, "**" can be used to denote "don't care" for Booleans as for any other data type. However, "--" or "Unspecified" can be used only for enumerated data types.

String Operator

There is one string operator, which performs concatenation of two strings to yield one resulting string. Concatenation may be used, for example, in the creation of tailored messages, as in:

```
Let outstr be "WARNING: " $ message $ " PLEASE RESPOND (Y/N): ".
```

In this example, the string variable "outstr" receives a string containing a variable "message", along with standard prefix and suffix strings.

Mathematical Notation (Preferred Form)	"English-like" Notation	Meaning
$	concatenated with	Concatenation

Only string values may be concatenated together. The result is a string consisting of the first string followed by the second string. Example:

```
"This is " $ "a test."
"This is "$"a test."
"This is " concatenated with "a test."
```

are all equivalent to:

```
"This is a test."
```

There is a way to include values of other data types by converting them to strings with the Evaluate statement as described under "Values and Simple Expressions" at the beginning of this section.

Order of Precedence

Whenever there is any ambiguity or uncertainty, parentheses should be used to specify the order in which operators should be applied within an expression. When more than one operator is used in a sequence, precedence relations are used to resolve the order. Operators with higher precedence are performed first; within the same precedence, operators are performed within the expression from left to right. Operators of the same precedence associate to the left. For example,

(a & b & c) = ((a & b) & c).

The following table gives the precedence of RAND-ABEL operators. Operators in the same row are of equal precedence.

Highest precedence:	~ (not)		- (unary)	
	*	/	modulo	
	+	-		
	<	>	<=	>=
	=	~=		
	& (and)			
Lowest precedence:	| (or)			

V. DECLARATIONS

TO DECLARE A VARIABLE

```
+-declaration--------------------------------------------------+
|                                                              |
|    Declare variable-name:                                    |
|    Declare variable-name by example:[1]                      |
|                                                              |
|        Let variable-name be expression.                      |
|        Let variable-name be identifier constant.             |
|        Let variable-name be enumerated variable.             |
|                                                              |
+--------------------------------------------------------------+
```

Rules:

1. The type of the variable becomes the same as the type of the expression.

2. The type of the expression must be uniquely determinable at the time this statement is encountered. (For example, if the same identifier constant appears in the range of several enumerated data types, then it may not be used in an assignment within a declaration.) If Type-color includes Red, Blue, and Green, while Type-mood consists of Happy, Blue, and Querulous, then

   Declare tint: Let tint be Green.

   declares the variable "tint" to be of type "Type-color", but

   Declare tint: Let tint be Blue.

   is ambiguous and therefore an error.

   Examples:

   ```
           Declare troop-strength:
               Let troop-strength be 10000.

           Declare force-ratio:
               Let force-ratio be 5.8.

           Declare message:
               Let message be "Help!".
   ```

[1]The phrase by example is optional in a declaration.

```
Declare agreement:
    Let agreement be Yes.

Declare current-force-test by example:
    Let current-force-test be Function calcl.

Declare alliance-member:
    Let alliance-member be France.
```

The spatial alignment of these statements is not important; they are aligned by variable name here merely for ease in reading.

TO DECLARE AN ARRAY

The syntax diagram below shows how to declare a RAND-ABEL array.

```
+-declaration---------------------------------------------------+
|                                                               |
|    Declare array-name:                                        |
|    Declare array-name by example:                             |
|                                                               |
|       Let array-name of simple-expr ,       . . .             |
|       Let array-name in simple-expr , and                     |
|       Let array-name by simple-expr , and                     |
|                                                               |
|                                  . . .  , simple-expr         |
|                                  . . .  , and simple-expr     |
|                                                               |
|                           be expression.                      |
|                                                               |
+---------------------------------------------------------------+
```

Rules:

1. There must be at least one *simple-expr*.
2. The type of the array is the type of the *expression*, which must be determinable at the time this statement is encountered.
3. Arrays can have one or more indices. (Arrays with zero indices are equivalent to variables.)

4. The *simple-exprs* that are used to index the array must be either of type integer or enumerated.

5. If any index is of type integer, it is designated by a single integer constant, n, in place of *simple-expr*. This index can then take on the integral values 0 . . . n . Note that index n means that the index can take on n+1 distinct values.

Note that a one-dimensional array, indexed by the smallest positive integers (1, 2, 3, . . .) is often called a "vector" in some other computer languages. A two-dimensional RAND-ABEL array indexed by the smallest positive integers corresponds with the term "matrix" in other computer languages.

Example of an array with integer indices and enumerated value:

```
Declare chessboard-square:
    Let chessboard-square of 7, and 7 be
        Type-chess-piece.
```

where Type-chess-piece has been previously defined as

```
Define Enumeration chessboard-square:
        king, queen, knight, bishop, rook, pawn, empty.
```

TO DECLARE A FUNCTION

Every function that is used must be declared. Every function either always returns a value, or never returns a value. The function declaration indicates which of these cases applies, as well as the data type of the arguments and value returned, if any.

```
+-declaration-----------------------------------------------+
|                                                           |
|   Declare func-name:                                      |
|   Declare func-name by example:                           |
|                                                           |
|        Let expression be Report from named-function-call. |
|        Perform named-function-call.                       |
|                                                           |
+-----------------------------------------------------------+
```

Rules:

1. The first form must be used when the function returns a value. The type of the *expression* must be the same as the type of value returned by the function. The type of the *expression* must be determinable at the time this statement is encountered.

2. The second form is used only when a function does not return a value.

3. A function must be declared before it is defined, and it must be defined before any use. See Sec. VI for a description of function definitions.

A *named-function-call* is one that explicitly uses the function name to invoke it, not a pointer to that function. See Sec. VI for a description of *named-function-call*.

Examples:

```
Declare select-country:
    Let France be Report from
          select-country using alliance as range,
                              and strength as criterion.

Declare force-calc by example:
    Let 5.0 be Report from force-calc using
                              France as country.

Declare validity-check:
    Perform validity-check.
```

Functions may have parameters associated with their use. Each such parameter is given a keyword that is used in the declaration of the function, in its definition, and in all calls to the function. The pairing of this keyword with a value means that arguments to a function can be listed in any order. Within the definition itself the keyword behaves like a local variable that has been assigned the associated argument value from the function's call. Such keywords must be unique for a given function, but can be (and frequently are) reused for other functions.

VI. FUNCTIONS

DEFINING A FUNCTION

RAND-ABEL has two types of functions: those that return a value (always), and those that do not (ever). The function declaration indicates which. A *declaration* is any of the declaration types (starting with the keyword Declare) listed in Sec. V titled Dec'ırations. It is an essential part of the function definition.

```
+-function-definition-----------------------------------------+
|                                                             |
| Define named-function-call : declaration . . .              |
|                              declaration                    |
|                                statement . . .              |
|                                statement                    |
|                                                             |
| End.                                                        |
|                                                             |
+-------------------------------------------------------------+
```

If the function returns a value, at least one of the statements within the function definition must be "Exit Reporting *simple-expr*". Moreover, one such statement must be reached during execution of the function, otherwise a run-time error will occur.

If the function does not return a value, it is exited either by an explicit Exit statement or else by "falling through" the *statements* to the End statement.

Local variables may be declared after the definition heading of a function and before any executable code. These, along with any function parameters (which are NOT declared in the function heading), may be referenced like any other variable throughout the function body but are not accessible to any other functions called from within that function. A function may call itself, either directly or indirectly, but is given a new set of local variables each time.

Note that local variables may also be declared after the opening brace and before any executable code within an internal program block. Such a variable will be valid until the corresponding closing brace is reached.

Examples:

```
Define Timed-wakeup:
    If Time is at least Time-to-wake of (Command-id of self)
    Then
    ( Record "Starting move at maximum time = " Time ".".
      Exit Reporting Yes.
    )
    Else Exit Reporting No.
End.
```

NAMED FUNCTION CALLS AND FUNCTION INVOCATIONS

A *named-function-call* is an invocation of a function in which the name of the function appears explicitly. It is required, for example, as part of the declaration of that function (which announced the names and data types of its arguments, and the type of its returned value, if any).

```
+-named-function-call-----------------------------------------+
|                                                             |
|  func-name                                                  |
|                                                             |
|  func-name  using expression as    param-name , . . .       |
|                              for             , and         |
|                                                             |
|                                                             |
|                    . . . ,      expression as  param-name  |
|                         , and              for             |
|                                                             |
+-------------------------------------------------------------+
```

Rules:

1. In a *named-function-call*, the *func-name* must be given explicitly; a pointer to a function is not allowed in this case.

2. When used as an example in a function declaration, the types of each *expression* must be determinable at the time the declaration is encountered.

By contrast, a *function-invocation* has the same form as a *named-function-call*, but it can have a pointer to a function in place of an explicit function name:

```
+-function-invocation-------------------------------------------+
|                                                               |
|  named-function-call                                          |
|                                                               |
|  func-ptr                                                     |
|                                                               |
|  func-ptr   using expression as    param-name ,       . . .   |
|                              for               , and         |
|                                                               |
|                   . . . ,     expression as  param-name        |
|                        , and             for                   |
|                                                               |
+---------------------------------------------------------------+
```

When a function does not return a value, it is invoked through the statement

Perform *function-invocation*.

When a function returns a value, it is accessed via the expression

Report from *function-invocation*

Examples:

```
Let message be Report from next-move
                using pawn as whites-last-move-piece.

If Report from Timed-wakeup
         using now as time is Yes
Then Perform Work.
Else Perform Error-handler.
```

If *function-to-perform* is a function pointer, then

```
Let function-to-perform be the Function next-move.
Perform function-to-perform using pawn as
  whites-last-move-piece.
```

VII. RAND-ABEL STATEMENTS

Statements are used in RAND-ABEL to define the operation of a function. (Several statements can also occur at the "top level" in RAND-ABEL outside of a function definition to set the global context in which other statements will operate: namely, declarations, the Data Dictionary (see Sec. IX), and the Trace and Untrace statements.

The various forms of RAND-ABEL statements are described below within the following categories:

Assignment
Conditional Execution
Repetitive Execution
Table Statement
Functions: Invoking and Exiting
Input/Output
Compound and Null Statements

All RAND-ABEL statements begin with a keyword that uniquely identifies the statement type. In general all RAND-ABEL statements end with a period; the only exceptions are compound, conditional, and repetitive statements whose form has an embedded *statement* as the last entity within the form; in those cases, the period ending the embedded *statement* becomes the statement delimiter.

ASSIGNMENT

Assignment statements are used to store a value into the location specified by either a variable or an array element.

```
+-statement------------------------------------------------------+
|                                                                |
|  Let lvalue be expression.                                     |
|  Let pointer be expression.                                    |
|                                                                |
|                                                                |
|  Increase  lvalue by expression.                               |
|  Decrease lvalue by expression.                                |
|  Multiply lvalue by expression.                                |
|  Divide lvalue by expression.                                  |
|                                                                |
+----------------------------------------------------------------+
```

Rule: The (data) types of the terms on the "lefthand side" and "righthand side" of the assignment statement must match.

Two exceptions:

1. If a real number is required by the lefthand side, then if the value of the *expression* is integer that integer is coerced into a real for the purpose of this statement.

2 The righthand side can be an enumerated identifier constant of ambiguous data type if that ambiguity is resolved by the type of the *lvalue* or *pointer* on the lefthand side. For example, the "Blue" in

   Let color be Blue.

   could be "mood" except that the type associated with "color" unambiguously identifies the type of Blue as a member of the enumerated data type "Type-color".

The terms *lvalue* and *expression* are defined in Sec. IV. In general, an *lvalue* is what can normally occur on the lefthand side of an assignment statement: namely, a term giving the address of a named storage location, not a pure value.

Examples:

```
Let gross-profit be gross-sales - cost-of-sales.

Let force-ratio be Report from force-calc
        using France as side-1 and Yugoslavia as side-2.

Decrease force-ratio by 2.5 .
```

CONDITIONAL EXECUTION

Conditional execution is controlled by the If statement. It allows certain RAND-ABEL statements to be executed only if certain conditions are true, or are false.

```
+-statement-----------------------------------------------------+
|                                                               |
|  If Boolean-expression Then statement                         |
|                                                               |
|  If Boolean-expression Then statement Else statement          |
|                                                               |
+---------------------------------------------------------------+
```

Rules:

1. The *Boolean-expression* is any expression that evaluates to type Boolean (i.e., that takes on values Yes and No).

2. If the *Boolean-expression* evaluates to Yes, then the first *statement* is executed.

3. If the *Boolean-expression* evaluates to No and the Else clause is present, the *statement* following the Else keyword is executed. If the *Boolean-expression* is No and no Else clause is present, no action is taken.

4. As is normal programming language practice, if conditional statements are nested, an Else clause is attached to the nearest previous If clause that does not yet have an Else clause attached. (If one needs a null If or Else clause to keep the logic straight, use the RAND-ABEL null statement, consisting of just a period, as in:

```
If king-unchecked Then.  Else Perform Think.
                        or
If king-in-check Then Perform Think.  Else.
```

This statement is not delimited by a period for reasons given at the beginning of this section: the last *statement* embedded within the If statement will contain its own delimiter.

Either *statement* can of course be a compound statement (that is, one or more *declarations* and *statements* contained within "(" and ")" braces) thereby allowing any needed c mplexity in logic to be stated in the Then or Else clauses.

Example:

```
If user-response is "Y" or user-response is "YES"
Then
( Perform Recalculation.
  Print   "Calculation Completed.  More? (Y/N): "
  Let user-response be Report from query-user.
)
Else If user-response is "N" or user-response is "NO"
     Then
     ( Print "No action taken.  More? (Y/N): "
       Let user-response be Report from query-user.
     )
     Else If  user-response is "?"
          Then Perform Help-function.
          Else
          ( Print "Your response not understood."
            Perform Help-function.
          )
```

Note that If . . . Then rules can also be formed using the table statement.

REPETITIVE EXECUTION

The RAND-ABEL For and While statements allow one or more statements to be executed repetitively--that is, zero or more times, depending on the controlling variable or expression.

```
+-statement-----------------------------------------------------+
|                                                                |
| For variable : statement                                       |
|                                                                |
| While Boolean-expression : statement                           |
|                                                                |
+----------------------------------------------------------------+
```

Rules:

1. In the first form, the *variable* must be of enumerated data type. The *statement* is executed once for each identifier constant in the range of the *variable*, with the *variable* bound in turn to each identifier constant, in the order in which the identifier constants are declared as being the range of the enumerated data type.

   Examples:

   ```
   Define Enumeration Type-alliance:  France, Germany, Spain.
   Declare alliance-members:
       Let alliance-members be Type-alliance.

   For alliance-members Perform Force-calc.

   For each-country (US or UK or FRG or Belgium):
   {
          Let Membership of each-country be Nato.
          Let Side       of each-country be Blue.
   }
   ```

   Note that the use of the or keyword here limits the execution of this for statement to only the listed identifier constants within the enumerated data type "each-country". All other elements of that data type are excluded by use of the or construct.

2. In the While form, the *Boolean-expression* is evaluated; if its value is Yes, then the statement is executed; if the value is No, no further action is taken. If the statement executes, the *Boolean-expression* is then re-evaluated, and if Yes the statement is re-executed. This sequence continues until the value of the *Boolean-expression* becomes No.

Example:

```
        Let k be 3.
        While k>0: (
                    Print resultsfile k.
                    Decrease k by 1.
                   )
```

leads to the following records sent to the resultsfile:

```
                    3
                    2
                    1
```

The following two RAND-ABEL statements are used within a repetitive execution to change the flow of the program's logic:

```
+-statement--------------------------------------------------+
|                                                            |
|   Continue.                                                |
|                                                            |
|   Break.                                                   |
|                                                            |
+------------------------------------------------------------+
```

Rules:

1. Within a repetitive execution, the Continue statement acts as completion of the current repetition, and control passes to the next repetition of the loop, if any.

2. The Break statement acts as completion of all repetitions of the loop, and control passes to the statement following the repetitive statement.

3. In both cases, control returns to the most immediately inclusive Table, For, or While statement. That is, to the innermost repetitive statement if they are nested.

   Examples:

```
        Let k be 3.
        While k>0:
        {
            If k=1 Then Break.
            Else {Print resultsfile k. Decrease k by 1.}
        }
```

leads to the sequence of records in resultsfile:

```
3
2
```

```
Let k be 3.
While k>0:
(
    If k=2 Then Continue.
    Else (Print resultsfile k. Decrease k by 1.)
)
```

leads to one printed record in resultsfile:

```
3
```

followed by an infinite loop, with k=2 and the While statement repetitively executing with no effects.

Repetitive execution can also be achieved by the RAND-ABEL table statement. This special RAND-ABEL statement is described in the following section.

TABLE STATEMENT

The Table statement is the most powerful statement in RAND-ABEL. It can be used to call a function repeatedly, with different arguments, or as a decision table. It is an example of a statement with a two-dimensional syntax; the spatial layout of the table-header is important in determining the meaning of the table statement.

```
+-statement-------------------------------------------------+
|                                                           |
|    Table func-name                                        |
|    Table compound-statement                               |
|    Decision Table                                         |
|                                                           |
|              table-header.                                |
|                                                           |
|              table-body.                                  |
|                                                           |
+-----------------------------------------------------------+
```

Function Table

Basically, the table statement allows the named function or the *compound-statement* to be executed once for each row of data in the *table-body*. If the table statement contains a named function, then the columns of data within the *table-body* are matched up with the function's formal parameters by means of the column headings within the *table-header*; if the table statement contains a *compound-statement*, then the local variables declared within the highest level block of that compound statement are matched with the columns of data within the *table-body* by means of the column headings within the *table-header*.

The concept and power of the table statement is best illustrated by example. The following RAND-ABEL table uses the function "Deploy". It is similar to the deployment table shown in Sec. I of this manual but, here, is expanded to seven columns.

```
Table Deploy
  [This table initiates the deployment of assigned forces to the
   Central European theater]

     qty     #-%  unit-type  unit-owner      assigned-to
     ======  ===  =========  ==============  ==============
                             in-area         to-area
                             ==============  ==============.
        100   %   Troops     Denmark         CEur
                             All             CEur-1
        100   %   Troops     Netherlands     CEur
                             All             CEur-2
         25   %   Troops     FRG             CEur
                             All             CEur-3
        100   %   Troops     UK              CEur
                             All             CEur-4
        100   %   Troops     Belgium         CEur
                             All             CEur-5
```

This table statement causes the function Deploy to be called five times, once for each row of the *table-body*. (Each row has seven entries, the last two being "folded over" so that they appear underneath the columns labeled "unit-owner" and "assigned-to".)

Decision Table

Another important use of a table statement is as a decision table. It is required by the syntax of the table statement that slash (/) be used within the table header of a decision table to separate the conditions from the action to be taken. For example, consider the following decision table (a macro table) used as the "Then" clause of a conditional statement:

```
If Current-situation is Eur-demo-tac-nuc

   ["Eur-demo-tac-nuc" represents the situation that one or both
    superpowers have used some tactical nuclear weapons in Europe,
    but have done so primarily for demonstrative purposes--i.e.,
    to coerce the opponent into terminating]

Then
{
  Table
  {
     Declare Basic-status#: Let Basic-status# be Basic-status.
     Declare Risks#:        Let Risks#        be Risks.
     Declare Escalation-guidance#: Let Escalation-guidance#
       be Escalation-guidance.

       If (Basic-status# is Basic-status or
            Basic-status# is Unspecified) and
          (Risks# is Risks or Risks# is Unspecified)
       Then
       {
            Let Escalation-guidance be Escalation-guidance#.
            Break.
       }
  }
                                                  Escalation-
                     Basic-status#      Risks# /   guidance#
                     ==============     ====== /  ==============

                     goals-met          --        Eur-term
                     progress-good      low       Eur-demo-tac-nuc
                     progress-marginal  low       Eur-gen-tac-nuc
                     progress-good      marginal  Eur-demo-tac-nuc
                     progress-marginal  marginal  Eur-gen-tac-nuc
}
```

Note the use of the Break. statement within the compound statement defining the operation of the Table statement, in order to stop the iteration through the table rows as soon as a satisfactory condition is found.

Using the Decision Table construct, the macro table above reduces to the succinct:

```
Decision Table
                                         Escalation-
Basic-status          Risks      /        guidance
================      =======   /      ================.

goals-met             --               Eur-term
progress-good         low              Eur-demo-tac-nuc
progress-marginal     low              Eur-gen-tac-nuc
progress-good         marginal         Eur-demo-tac-nuc
progress-marginal     marginal         Eur-gen-tac-nuc
```

The rules for constructing a *table-header* are as follows:

1. A *table-header* consists of one or more "text island," each representing the name of a parameter (if a function is named) or the name of a local variable (if a *compound-statement* is used) or the name of a local or global variable (in decision tables).

2. A "text island" is a two-dimensional grouping of characters such that each character of the group is directly adjacent (either horizontally or vertically--not diagonally) to some other character in the group.

3. Spaces are not permitted within an identifier used as a column header, so indications of ownership in such an identifier (e.g., "Red's Presumed-opponent") are not permitted.

4. Newlines are checked for in tables at the end of each logical row. There may be extra newlines interspersed (allowing the multiline-per-row table as shown), but the newline break at the end of the row must occur. This is quite useful for finding errors involving table rows with missing or extra items.

5. In addition to normal "white space" characters (space, tab, newline) and comments (enclosed in square brackets), the equal sign (=) is also considered "white space" in determining the "text islands" composing a table header.

6. If a "connector character" is useful in retaining the integrity of a "text island," the following characters may be used. They provide the adjacency required by rule #2 above, but are not themselves considered part of the identifier represented by the text island:

   | (   )

   (These connector characters also "count" in determining the ordering of the text islands; that is, their position as part of a column heading helps determine the relative position of that column heading.)

The following example is a table header contrived to demonstrate most of the above rules:

```
Table Red-to-3rd-countries

  country-
     (             cooper-          =european-======swa-====
    affected  side  ation          =involvement==involvement=
  ==========  ====  =============  ============================
  France      White Uncooperative  Disengaged      Disengaged
  GDR         Red   Combat-basing  On-Call         Noncombatant
```

This example calls the function "Red-to-3rd-countries" twice (once for each row of the table). The data in the table body are matched to five function parameters having the following names: country-affected, side, cooperation, european-involvement, swa-involvement. Those function parameters need not have been declared or defined in that order.

The use of the vertical bar (|) as a connector character keeping a "text island" together within a *table-header* allows text headings to be associated with individual columns of a table in a very flexible way. Consider the following valid RAND-ABEL table statement:

Table Initialize

```
               Country-set [is it a country?, not a region/sea]
               |
               | Superpower-set [is it one?]
               | |
               | | Player-status [should the model simulate it?]
               | | |                                                 Decision
               | | | Borders-WP                                       -delay
               | | | |                                               [1-366]|
               | | | | Assertive-country [always fight if attacked]  [ days]|
               | | | | |                                                    |
               | | | | | Nuclear-capable Mem  Orien           Red-  Blue-   |
               | | | | | |               ber  ta    Tempera   pres  pres    |
Region         | | | | | | Leader        ship tion  ment      ence  ence    |
============== = = = = = = ============= ==== ===== ========= ===== =====  ==
Afghanistan    Y N Y Y N N USSR          --   Red   Captive   Major None    1
Arabian-Sea    N N N N N N --            --   --    --        --    --     --
Australia      Y N Y N N N UK            ANZS Blue  Moderate  None  Token   2
Austria        Y N Y Y N N --            --   White Reluctant None  None    4
Belgium        Y N Y N N N US            NATO Blue  Reliable  None  TripW   1 .
```

At times, more table columns are needed to describe a situation than will fit in the width of a single page. To allow wide tables to be described, the following additional format rules for a *table-header* allow a "wrap-around" header to be created, in which one or more additional rows of "text islands" provide the needed continuation.

Rules for constructing a multirow *table-header*:

(1) Table headers may be continued onto succeeding lines, if all characters in text islands composing one row of the header are below all characters composing the previous row of the header.

(2) Within a row of text islands, column headers are read left-to-right.

(Note that by this set of rules, the table above qualifies as having only a single row of column headers, since the vertical bars (|) associating text strings with columns keep the "text islands" defining each column header from being separated vertically.) In table construction, the keyword -- can be used as a synonym for **Unspecified**, and ** is used to indicate "don't care".

Multirow table headers are best understood by example. Consider the following table statement, with six formal parameters:

```
  Table Function-of-6
==================================================================
First-          Second-          Third-          Fourth-
 parameter        parameter        parameter        parameter

         Fifth-           Sixth-
          parameter        parameter
==================================================================
  12.5             Green             512          "String 1"
          10002             (A + 10)
   9.0             Blue              221          "String 2"
          9943              (A - 24)
```

Note that a blank line has been used to clearly separate the first and second rows of the table header; this is not strictly necessary, but aids in keeping the text islands separate. Note also that the entries in the table body "wrap around" in the same manner. In fact, the entries are merely read line-at-a-time and matched to the corresponding headers in the table header. Although they have been staggered so that they may be placed beneath their corresponding header, this is again not strictly necessary; it simply aids in human comprehension of the table.

The rules for a *table-body* are simple: A *table-body* consists of a sequence of entries, each of which is a *unitparam*. (See the section Values and Expressions for the formal definition of a *unitparam*. It is essentially a primitive value or a parenthesized expression.) The following additional rule holds for a *table-body*:

Rules:

1. If the *table-header* describes n formal parameters or variable-names, then the number of entries in the table body must be a multiple of n. (Normally the entries are placed in columns beneath the column headers within the *table-header*, so that each row of the table naturally consists of n entries, except when wide rows "spill over" onto the next line as in the example above.)

2. Each entry must match in type with the corresponding formal parameter or local variable. For example, "Second-parameter" is an enumerated data type of type Type-color.

The *table-header* and *table-body* are each followed by a period (.) as delimiter.

The RAND-ABEL Translator that interprets a table merely counts n entries in the table body, then either calls the named function or executes the *compound-statement*, then acquires the next n entries (until a "." is encountered instead). There is no meaning attached to the grouping of table entries into rows.

FUNCTIONS: INVOKING AND EXITING

The declaration of functions was covered in Sec. V (Declarations). Section VI is devoted to the definition of functions and presented the syntax for a *function-invocation*. Functions returning a value are invoked by the expression Report from *function-invocation*. Functions not returning a value (presumably executed for their side-effects) are invoked by a Perform statement, discussed here as part of a description of all RAND-ABEL statements. We also present here the Exit statement that allows completion of a function's execution, whether or not it returns a value.

```
+-statement-----------------------------------------------------+
|                                                               |
|   Perform function-invocation.                          |
|                                                               |
+---------------------------------------------------------------+
```

Rule: This statement is used to execute a function that does not return a value. (That is, it is executed for the side-effects it causes.)

In some programming languages, a function not returning a value is called a subroutine. In RAND-ABEL, all program logic is contained in functions; a function not returning a value is equivalent to a subroutine.

Example:

```
        Perform force-ratio-calc
                using France     as side-1,
                      Yugoslavia as side-2,
                  and  3.5 E 4   as multiplier.
```

Within the statements defining a function, the following statement is used to return program control to the place from which the function was invoked:

```
+-statement-----------------------------------------------------+
|                                                               |
|   Exit.                                                       |
|   Exit Reporting simple-expr.                                 |
|                                                               |
+---------------------------------------------------------------+
```

Rules:

1. If the Reporting clause is omitted, then the function does not return a value.

2. If the Reporting clause is used, then the function always returns a value of the same data type as the *simple-expr*.

3. If there is more than one Exit Reporting statement within the definition of a function, then each of those statements must contain an *expression* of the same data type. However, the *simple-expr* may be of ambiguous data type if that ambiguity is resolved by the function's declaration.

If a function does not return a value, it is always invoked by the RAND-ABEL statement:

Perform *function-invocation*.

If a function returns a value, then it is always invoked using the expression:

Report from *function-invocation*

Even functions reporting a value may have side-effects and in that sense are not equivalent to a mathematical function. If a function is declared as returning a value, then it must return a value using the Reporting clause. If it is not declared as returning any value, it must return using Exit.

Examples:

```
Exit.
Exit Reporting "Success."
Exit Reporting ((multiplier * force-ratio)/2.0).
```

INPUT/OUTPUT

Print, Log Statements

The following I/O statements are used to communicate with the "outside world"--that is, the computer system environment within which RAND-ABEL is running.

Before the formal syntax description, some general terms should be understood by the reader. RAND-ABEL operates within a C language environment, within the UNIX operating system. The general characteristics of C and UNIX are assumed. The UNIX system has the (very powerful) concepts of "standard input" (which is often a terminal's keyboard) and "standard output" (which is often a terminal's display screen). Input and output consist of a stream of characters, which is usually directed to the standard input and output ports. However, these data streams can be redirected, for instance into a file, or into the output or input streams of another process running in the computer.

As data are emitted from a RAND-ABEL program, it is either formatted (according to its data type) in a standard (i.e., default) manner, or else the programmer can exercise some control over the format in which it appears. A special language of format codes, consisting of a string of characters, is used to specify formatting of I/O. The default formatting for each of RAND-ABEL's data types, and the special format codes, are described within this section.

```
+-statement----------------------------------------------------------+
|                                                                    |
|   Print unitparam  . . .  unitparam.                               |
|   Print with format-spec unitparam  . . .  unitparam.              |
|                                                                    |
|   Print streamname unitparam  . . .  unitparam.                    |
|   Print streamname with format-spec unitparam . . . unitparam.     |
|                                                                    |
|   Log unitparam  . . .  unitparam.                                 |
|   Log with format-spec unitparam  . . .  unitparam.                |
|                                                                    |
|   Log streamname unitparam  . . . unitparam.                       |
|   Log streamname with format-spec unitparam . . . nitparam.        |
|                                                                    |
+--------------------------------------------------------------------+
```

Rules:

1. The Log statement causes a stream of data (defined by the sequence of *unitparams* within the statement to be sent to the output stream.

2. The Print statement causes a stream of data (defined by the sequence of *unitparams* within the statement) to be sent to an output stream.

3. The default output stream is the UNIX *stdout*; if a *streamname* is given, output from Print or Log is directed to that output stream instead.[1]

4. If the optional with *format-spec* clause is omitted, all output is formatted according to standard defaults determined by the data types of the *unitparams* being output.

5. If the with *format-spec* clause is included, the *format-spec* is a RAND-ABEL expression of type string. The character string is interpreted as a specification for formatting of output, and output is formatted according to its specifications.

6. A value of type pointer can be output as a hexadecimal number for debugging purposes, but this is not expected to be used in a production program.

[1]If the RAND-ABEL program is executing in the context of the RSAC system, output should not be sent to the default *stdout*, as this will conflict with system CRT screen management.

The definition of a *unitparam* is contained in the subsection Values and Expressions within Sec. IV. Basically, it is a simple value or a parenthesized RAND-ABEL expression.

Default output formats are used for each data type when no control is provided by an explicit *format-spec*. These default output formats are described in the following subsection.

Examples of the Print and Log statements are given at the end of this subsection, after the various formatting options are presented.

Default Output Formats

If no special format controls are given, each data type has a standard way in which its value is printed. These default output formats are given by the following table.

Data Type	Default Output Format
Integer	A string of digits, with an optional prefix minus sign. No decimal point. Delimited by one blank on each side.
Real	A string of digits with an embedded decimal point. At least one digit is printed before and after the decimal point, even if it is a zero. Optional prefix minus sign. Numbers less than one-millionth (1 E -6) are considered a zero. Otherwise, for numbers less than one, enough decimal places are printed to show at least two digits of significance. Delimited by one blank on each side.
String	The string is printed literally, with no surrounding quotation marks, and not delimited by blanks.
Boolean	The string "Yes" or "No" is printed, without surrounding quotation marks. Delimited by one blank on each side.
Pointer	A pointer-type value is output as a hexadecimal number for debugging purposes,

	using the same conventions as the integer data type.
Enumerated	The identifier constant is printed without surrounding quotation marks. Delimited by one blank on each side.

Format Specification

A *format-spec* is used to control the formatting of output. It is a sequence of characters that are printed as listed, except when the special characters "%" and "\" are encountered. The % is followed by a special formatting code. The formatting codes recognized by RAND-ABEL are as follows:

%i	Enumerated data type
%s	String
%b	Boolean (Yes or No)
%d	Integer without a decimal point
%e	Real (or floating point) scientific notation
%f	Real (or floating point) fixed decimal point
%g	Real (or floating point) general; uses scientific notation or fixed decimal point, whichever is shortest.

A literal percent sign is entered in a format string as %%.

A number may be placed between the percent sign and the letter. That number specifies the overall number of characters allocated to the value. If a number is used, the field will be blank padded, unless the field width number begins with a leading zero, in which case the field will be zero padded. The field width number can optionally be followed by a decimal point, and then another number. The second number will be the number of digits to appear after the decimal point for %e, %f, or %g formats.

There are other special options that can be used in these format strings. They obey the conventions of "printf(3)" in Section 3 of the *UNIX Programmer's Manual*, Bell Laboratories. That document should be consulted for more detailed information.

The backslash (\) is followed by a character or sequence of 3 octal digits that represent special characters:

\n	Newline (line feed)
\r	Carriage return
\t	Horizontal tab
\b	Backspace
\f	Formfeed
\\	Backslash
\'	Single quote
\ddd	Any bit pattern (exactly 3 digits in octal notation)

These special escape sequences allow any ASCII character to be produced. For example, "\n" allows more than one line of text to be put in the same string, and "\f" causes a page eject. By using a 3-digit octal (i.e., base 8 number system) code, any ASCII character can be produced; e.g., one could make the CRT terminal "bell" ring by the following statement:

```
Print "\007".
```

Streams

A stream is a pathway through which information is transferred from a program to a terminal, file, or other program. The information is transferred as a stream of characters.

The normal output stream for Log and Print statements is the UNIX "standard output," which is initially set to the user's terminal.

If a *streamname* has been used in a Log or Print statement, but that stream has not yet been opened, then a runtime error will be generated.

The three predeclared and preopened streams are: "Input", "Output", and "Error".[2] "Input" is the stream of characters received from the user's terminal keyboard. Reading a character from Input causes UNIX to wait for the user to type in a line of input text.

[2]It will help the C programmer to know that these correspond directly to C's *stdin*, *stdout*, and *stderr*.

"Output" corresponds to the user's terminal screen. Printing a line on the stream "Output" causes the line to appear on the user's terminal. "Error" is also directed to the user's terminal. It is defined separately from "Output" since the program may want to redefine one of these to go somewhere else.

RAND-ABEL supports three predefined, stream-oriented functions. They are:

Function Name	Argument 1	Argument 2	Return Value
Open-stream	*file-name*	*mode*	*streamname*
Close-stream	*streamname*		(none)
Flush-stream	*streamname*		(none)

In the above function calls, *file-name* is a string argument that is either a UNIX file name or else a full UNIX pathname (i.e., giving directory, subdirectory, etc.). *Mode* is one of the strings: "read", "write", "append". The value returned from the Open-stream function should be assigned to an integer variable that stores the ID of the stream. This same variable is then used as an argument to the Close-stream and Flush-stream functions.

The Open-stream function associates a UNIX file or path, in read, write, or append mode, with a *streamname*. If a file is opened in write mode and the file does not exist, it is created. If the file does exist, it is deleted first. If a file is opened in append mode, all writing to that file is appended to the end of the existing file, if any.

The Close-stream function closes a stream, making it unavailable for further use (until reopened). It is standard practice to close streams when they will no longer be used by the program.

The Flush-stream function is useful primarily for debugging purposes. Typically, when a RAND-ABEL program executes a Log or Print statement, the only effect is to fill that file's buffer in the operating system. Later, the operating system will perform the actual write to the file on disk. This buffering of output provides significant performance advantages. This buffer will be written to the appropriate file when a RAND-ABEL program stops execution in the normal manner. However, it is possible for a RAND-ABEL program that has an error to abnormally exit without first writing the buffer to disk. This can cause the programmer to think that his RAND-ABEL program terminated at a point much earlier than is actually the case. To get around this problem, the Flush-stream function can be called to write the contents of the buffer to disk. Because continued use of this function can degrade system performance, it is used primarily for debugging purposes.

Since "Input", "Output", and "Error" are already predeclared by the system, they can be used to declare other stream variables. For example:

```
Declare Output-file by example:  Let Output-file be Output.
```

Examples of Input/Output Statements and Functions

The following examples illustrate many of the possible uses of the various input/output statements and functions described in this section.

```
[ Declare Message-file to be a streamname ]

   Declare Message-file by example:  Let Message-file be Output.

[ Use the Open-stream function to associate a UNIX file with this
  streamname, and set its mode to write-only ]

   Let Message-file be Report from Open-stream
        using "~anderson/ABEL/programs/messages" for file-name,
          and "write" for mode.

[ Perform a set of writes to that file ]

   Print Message-file with "Threat level is now: %5d in country: %i\n"
                               threat (Report from select-country
                                        using Country-list as options).
   Print Message-file with "Force ratio is %3.1f at time %d \n"
                               ratio game-time.
   Print Message-file "End of game reached. \n\n"

[ Close file ]

   Perform Close-stream using Message-file as streamname.
```

COMPOUND AND NULL STATEMENTS

A compound statement is a sequence of zero or more declarations followed by a sequence of zero or more RAND-ABEL statements, all delimited by braces. It can occur wherever a *statement* can. It allows more complex program logic to be described than is allowed by the basic set of RAND-ABEL statements:

```
+-statement-----------------------------------------------------+
|                                                               |
|   { declaration . . . declaration                             |
|                                                               |
|        statement . . . statement  }                           |
|                                                               |
+---------------------------------------------------------------+
```

Rules:

1. All *declarations* occurring within a compound statement are local to that statement; they have no effect outside that statement.

2. Each *declaration* is processed in turn, then each *statement* is executed in turn. To obtain more control flow options, conditional and repetitive execution statements can be used, as well as function invocations.

Notice that a *function-definition* is not allowed within a compound statement. All function definitions are at the "top level" of a RAND-ABEL program.

The compound statement is not terminated by a period, since a period occurs as a delimiter to the last statement within its body.

If the compound statement requires more than one line of program text, it is traditional to line up the braces vertically, for ease in visualizing the matches between balancing braces. This positioning is for human consumption only; it is not used by the RAND-ABEL Translator.

Example: The following set of nested conditional statements uses compound statements to denote the set of RAND-ABEL statements to be executed at various places within the conditional logic. This example is repeated from earlier in this manual.

```
If user-response = "Y" or user-response = "YES"
Then
{ Perform Recalculation.
  Print  "Calculation Completed.  More? (Y/N): "
  Let user-response be Report from query-user.
}
Else If user-response = "N" or user-response = "NO"
     Then
     { Print "No action taken.  More? (Y/N): "
       Let user-response be Report from query-user.
     }
     Else If user-response = "?"
          Then Perform Help-function.
          Else
          { Print "Your response not understood."
            Perform Help-function.
          }
```

A null statement consists of a period, the normal terminating delimiter on a RAND-ABEL statement, only. It is useful within conditional statements to control logic flow. Empty curly braces are also a null statement.

```
+-statement---------------------------------------------------------+
|                                                                   |
|   .                                                               |
|                                                                   |
|   {}                                                              |
|                                                                   |
+-------------------------------------------------------------------+
```

Rule: This statement causes no effect.

Note that a side benefit of this statement is that extraneous periods used in error as statement delimiters (for example, after a conditional statement) do not cause a syntax error and have no effect.

Example: The following example, repeated from earlier in this manual, illustrates the use of the null statement to control logic flow within a conditional statement.

```
If king-unchecked Then.  Else Perform Think.
```

VIII. META-STATEMENTS

The following special statements can be used to influence how RAND-ABEL programs are written and interpreted.

#DEFINE

The **#define** statement provides an ability to create a macro giving a synonym or alias for a string of characters to be substituted wherever that macro identifier appears:

```
+-meta-statement---------------------------------------------+
|                                                            |
|   #define name [ unquoted-string ].                        |
|                                                            |
+------------------------------------------------------------+
```

Rules:

1. The *name* may be any RAND-ABEL *identifier*.
2. Wherever that identifier appears, it is replaced by the *unquoted-string* sequence of characters BEFORE THE RAND-ABEL TRANSLATOR INTERPRETS THE RESULTING STATEMENT.
3. After replacement, the RAND-ABEL Translator continues its scan at the beginning of the replacement string, so any **#define** identifiers it contains will similarly be replaced. **#define** statements may be nested to any level.

This form of "macro string substitution" can be used to change the surface appearance of RAND-ABEL programs. It should be used cautiously, since the resulting programs might well become less readable to persons who know the RAND-ABEL language.

Example:

```
        #define c-decl
           [ Declare country:
                Let country be France.
           ].
```

INCLUDE

```
+-meta-statement-------------------------------------------+
|                                                          |
|   Include "filename".                                    |
|                                                          |
+----------------------------------------------------------+
```

Rules:

1. The contents of the file whose name (or pathname) is given are inserted at this point in the RAND-ABEL (or C) program. The file name or pathname is interpreted relative to the UNIX directory containing the current source file.

2. After the text insertion takes place, the interpretation of the resulting file begins at the start of the newly inserted text lines, so if they contain Include statements, those statements are executed as they are encountered. Include files can be nested up to eight levels deep.

An Include statement is often used to incorporate a standard set of declarations or definitions into a RAND-ABEL program.

Example:

```
        Include "libraries/red-agent/dictionary.D".
```

DEBUGGING: TRACE AND UNTRACE

```
+-statement-------------------------------------------------+
|                                                           |
|   Trace If.                                               |
|   Trace Function.                                         |
|                                                           |
|   Untrace If.                                             |
|   Untrace Function.                                       |
|                                                           |
+-----------------------------------------------------------+
```

Rules:

1. Trace turns on reporting for either If statements or function invocations; Untrace turns off reporting.

2. All trace data are appended to a special file named "debug.out" within the current UNIX directory.

Function trace data consists of readable statements upon entrance to a function stating the function's name and the values assigned to each of its formal parameters. If the function returns a value, that value is reported to the file upon exit from the function.

"If" trace data writes to the same "debug.out" file. Each execution of a conditional statement, when Trace If is on, causes the conditional statement itself to be written to the file, along with an indication of whether the *Boolean-expression* evaluated to Yes or No.

Trace and Untrace are not executable statements. Rather, there are commands to the RAND-ABEL Translator to embed tracing information within the generated C program. Trace and Untrace statements can be nested; an Untrace turns off the corresponding nearest Trace of the same type.

Tracing can significantly reduce the speed of RAND-ABEL program execution and tends to generate large amounts of output. It should therefore be used selectively and only during program development.

Currently, the Interpreter allows two additional forms of tracing, namely Trace Decision Table and Trace Let.

IX. DATA DICTIONARY

The Data Dictionary facility in RAND-ABEL permits large, complex systems to be developed from separate modules that are created individually by different programmers. It is a much more elaborate and useful facility than the old concept of a "common" area in programs that stores data used in common by the different programs.

The RAND-ABEL Data Dictionary describes the contents and attributes of a data set to be used in common by all the RAND-ABEL modules constituting a system.[1] This common data set contains the specification of:

- A list of items (variables, attributes, tables, etc.) to be included in the common data set
- A list of items, similar to the list above but including subprocedures and functions, which are not part of the common data set
- A structuring of the source files that make up the system
- Access restrictions, ownership, method of implementation, and other such attributes for data items and source code
- Ancillary information, such as the author, module name, and other administrative attributes associated with data items and source code

The Data Dictionary consists of a set of files that are maintained in an extended RAND-ABEL language. The RAND-ABEL processor translates these files into C language data structures. The resulting C code can be used by a system monitor (a special program providing the foundation for the system being developed) for allocating memory for the data items described and also by a front-end data editor that can be used for display and manipulation of these items.

[1]In the RAND Strategy Assessment System (RSAS), this common data set is called the World Situation Data Set (WSDS).

A Data Dictionary entry begins with a Declare statement. (See the earlier section of this manual on Declarations.) After this declaration of an item are a number of statements describing the item. The sequence of descriptions is ended when a new Declare is found for the next item, a new Default statement is reached, or the End Declarations statement is encountered.

Many attributes that can be associated with a data item will be the same for an entire group of items (e.g., author, access restrictions, etc). To avoid the need for typing a whole list of attributes for each item, "default" attributes may be declared. When a default is declared, it affects all subsequently declared data items within the current file and any files "Included" within that file. A default does NOT affect any files that have "Included" the file that contains it. This nested-default mechanism allows higher-level files to create default environments for lower-level files without worry of a default in a lower-level file causing side-effects.

The set of Data Dictionary declarations has the following syntax:

```
+-data dictionary specification block----------------------------+
|                                                                |
|   Begin Declarations.                                          |
|   [No] Default DDdeclaration ...                               |
|                                                                |
|      declaration                                               |
|                                                                |
|             DDdeclaration . . .                                |
|                                                                |
|             DDdeclaration                                      |
|                                                                |
|      declaration                                               |
|                                                                |
|             DDdeclaration . . .                                |
|                                                                |
|             DDdeclaration                                      |
|                                                                |
|      . . .                                                     |
|                                                                |
|   End Declarations.                                            |
|                                                                |
+----------------------------------------------------------------+
```

The individual data dictionary declarations (i.e., *DDdeclarations*) are of three types:

1. *Defining declarations*. Information that actually affects the object code, such as type, size, or access data.
2. *Identifying declarations*. Information that is documentary but mandatory.
3. *Informative declarations*. Information that is optional but useful as documentation.

Each of these categories of declarations is described below.

DEFINING DECLARATIONS

These declarations are mandatory for each external data item but may be handled by default declarations that are in effect. (See Creating and Removing Default Declarations below.)

```
+-DDdeclaration-------------------------------------------------+
|                                                               |
|   Method: Direct.                                             |
|   Method: Function.                                           |
|   Method: Macro.                                              |
|                                                               |
|   Function: func-name.                                        |
|                                                               |
|   Macro: string-literal.                                      |
|                                                               |
+---------------------------------------------------------------+
```

Rule: Method means "method of access." An item's access method tells whether the variable is accessed directly or via a function or macro. If an item is accessed via a macro, the macro must be defined using a Macro statement. If it is accessed via a function, the name of the function must be given using the Function statement. The Macro or Function statement must immediately follow the Method declaration.

Examples:

    Method Function.
    Function: calculate-attrition.

where "calculate-attrition" must be a function that returns a value.

    Method Macro.
    Macro: ((*GOVERN_entry)(char *)F + 2) - F->governs + 1 .

```
+-DDdeclaration--------------------------------------------------+
|                                                                |
|   Use:  Clone.                                                 |
|   Use:  No Clone.                                              |
|   Use:  Constant.                                              |
|                                                                |
+----------------------------------------------------------------+
```

Rule: The Use declaration indicates whether an item is to be created dynamically (Clone) when the Push function is invoked and discarded when the corresponding Pop function is executed; or whether one instance of the data item is to be maintained throughout a Push and Pop (No Clone). Data that are never changed during program execution are declared with the Constant option.

Currently, No Clone is not implemented. Variables declared No Clone will behave as declared Clone.

Example:

    Use: Clone.

```
+-DDdeclaration--------------------------------------------------+
|                                                                |
|   Owner:  owner-name.                                          |
|                                                                |
|   Owner:  Global.                                              |
|                                                                |
+----------------------------------------------------------------+
```

Rule: This declaration allows different modules to have separate items with the same name. Source code also has an owner and automatically accesses either its own or "global" data items, unless otherwise specified by this declaration.

The "*owner-name*" is one of a set of commonly agreed-upon names by which the various groups developing code are identified. The keyword Global is used if there is no specific owner.

Examples:

```
                    Owner:  Red.
                    Owner:  Global.
```

```
+-DDdeclaration-----------------------------------------------+
|                                                              |
|   Read     Everyone.                                         |
|   Read     owner-name  . . .  owner-name.                    |
|   Read     owner-name , . . . , owner-name.                  |
|                                                              |
|   Noread   Everyone.                                         |
|   Noread   owner-name  . . .  owner-name.                    |
|   Noread   owner-name , . . . , owner-name.                  |
|                                                              |
|   Write    Everyone.                                         |
|   Write    owner-name  . . .  owner-name.                    |
|   Write    owner-name , . . . , owner-name.                  |
|                                                              |
|   Nowrite  Everyone.                                         |
|   Nowrite  owner-name  . . .  owner-name.                    |
|   Nowrite  owner-name , . . . , owner-name.                  |
|                                                              |
+--------------------------------------------------------------+
```

Rule: These declarations specify which source code owners (i.e., "access groups") can read or write this item. The "No" prefix can turn off a default or serve documentary purposes by establishing a lack of access for a particular group.

The special group Everyone applies to all access groups and can be used to grant or deny access for all groups.

Examples:

    Noread Blue, Neutral.
    Write Everyone.

```
+-DDdeclaration-----------------------------------------------------+
|                                                                   |
|   Read Format: string-literal.                                    |
|                                                                   |
|   Write Format: string-literal.                                   |
|                                                                   |
+-------------------------------------------------------------------+
```

Rule: The preferred format for reading and writing this data item is stated as a quoted string of format descriptors. (See the subsection Format Specification within the INPUT/OUTPUT portion of Sec. VII, RAND-ABEL Statements.)[2]

It is desirable to specify output formats for string variables, integers, and real (floating point) data whenever possible, since it helps the display programs format the data in a consistent manner. It is unnecessary to specify output formats for enumerated variables since the field width needed to display them is easily determined by the display programs.

Example:

    Write Format: "%5.3f".

[2]For programmers familiar with the C language: the format specification is the same as those used for the *scanf* and *printf* functions.

```
+-DDdeclaration-------------------------------------------------+
|                                                               |
|   Validation Range: numeral to numeral.                       |
|                                                               |
|   Validation Function:  func-name.                            |
|                                                               |
+---------------------------------------------------------------+
```

Rule: In the first form, an inclusive range of numeric values (either integer or real) is given. In the second form, a Boolean function is named that is expected to return Yes for a valid item, and No otherwise.

Examples:

```
        Validation Range: 2.7 to 8.75 .
        Validation Function: Check-value.
```

```
+-DDdeclaration-------------------------------------------------+
|                                                               |
|   Prompt Function:  func-name.                                |
|   Prompt String:    string-literal.                           |
|                                                               |
+---------------------------------------------------------------+
```

Rule: In the first form, the named function will be called prior to input for this data item. It returns a string that will be displayed on the user's CRT screen. In the second form, a quoted character string is given for display prior to input of the data item. In the RSAS environment, the input of data is performed from the Data Editor program.

Examples:

```
        Prompt Function: Show-message.
        Prompt String:    "Type ratio as a decimal: ".
```

```
+-DDdeclaration-------------------------------------------------+
|                                                               |
|   Initialize.                                                 |
|                                                               |
|   No Initialize.                                              |
|                                                               |
+---------------------------------------------------------------+
```

Rule: This declaration indicates whether the item can be initialized by the RAND-ABEL Translator or not.

At this time, the Initialize declarations may be used in a program, but their effect has not been implemented. Consequently, they do not alter program behavior.

Examples:

```
                    Initialize.
                    No Initialize.
```

IDENTIFYING DECLARATIONS

These declarations are mandatory but take an arbitrary comment as an argument. They are used for standardized documentation of a module.

```
+-DDdeclaration----------------------------------------------------+
|                                                                  |
|   Author: comment.                                               |
|                                                                  |
|   Date: comment.                                                 |
|                                                                  |
|   Definition: comment.                                           |
|                                                                  |
+------------------------------------------------------------------+
```

Rule: The *comment* may be a free-format comment enclosed in the documentation of the program or data item.

Examples:

```
                 Author: [ Mark LaCasse ].
                 Date:    [ 83/02/05 ].
                 Definition: [ This function returns a
                               string value that should
                               be displayed prior to input
                               of the force structure ratio ].
```

INFORMATIVE DECLARATIONS

These declarations are optional. They provide additional structured documentation of a RAND-ABEL program module.

```
+-DDdeclaration---------------------------------------------------+
|                                                                 |
|   References: comment.                                  |        |
|                                                                 |
|   Comments: comment.                                    |        |
|                                                                 |
|   Status: comment.                                      |        |
|                                                                 |
+-----------------------------------------------------------------+
```

Rule: The *comment* may be a free-format comment enclosed in square brackets "[ ]" that aids in the documentation of the program or data item.

The Status declaration is often used to represent whether a variable is "proposed" (indicating the author is willing to entertain proposals for change) or "confirmed" (indicating the author has closed debate on the subject).

Examples:

```
References: [ See R-1258, Section II ].
Comments: [ This function is a placeholder
            until a more complete computation
            can be developed ].
Status: [ Proposed ].
```

CREATING AND REMOVING DEFAULT DECLARATIONS

As mentioned above, all the mandatory Data Dictionary declarations need not be given for each data item or function. Many of these declarations can be covered by use of declared defaults.

Any of the *DDdeclarations* described in this section may be preceded by the keyword Default. If that is done, that setting for the particular *DDdeclaration* remains in force within the current file (and files Included within it) until a new default is given or a No Default is declared for that type of *DDdeclaration*. Any default setting may be overridden by individual *DDdeclarations* associated with a particular data item or function.

Examples:

```
        Default Owner:  Red-Agent.
        Default Method: direct.
        Default No Initialize.
        Default Author: [ Mark LaCasse, randvax!lacasse ].
```

The following No Default statements may be issued to remove a default setting on a type of *DDdeclaration*:

```
+-DDdeclaration-------------------------------------------+
|                                                         |
|   No Default Author.                                    |
|   No Default Comments.                                  |
|   No Default Date.                                      |
|   No Default Definition.                                |
|   No Default Initialize.                                |
|   No Default Method.                                    |
|   No Default Owner.                                     |
|   No Default Prompt Function.                           |
|   No Default Prompt String.                             |
|   No Default Read Format.                               |
|   No Default Write Format.                              |
|   No Default References.                                |
|   No Default Status.                                    |
|   No Default Use.                                       |
|   No Default Validation Function.                       |
|   No Default Validation Range.                          |
|                                                         |
+---------------------------------------------------------+
```

When No Default is specified for any mandatory *DDdeclarations*, such a *DDdeclaration* must accompany each data item or function until the next Default or End Declarations statement is reached.

Examples of No Default statements:

```
        No Default Comments.
        No Default Prompt String.
        No Default Read Format.
        No Default Validation Function.
```

EXAMPLE OF A DATA DICTIONARY DECLARATION SECTION

The following is a complete example of a Data Dictionary declaration section within a RAND-ABEL program. Such a RAND-ABEL code section is often contained in a file that can be included within another RAND-ABEL file to obtain the standard defaults and definitions required.

```
[ Sample Data Dictionary Entries for Blue Agent    January 1984 ]

Begin declarations.

   Default Owner:     Blue.
   Default Method:    Direct.
   Default Read:      Blue.
   Default Write:     Blue.
   Default Use:       Clone.
   Default No Initialize.
   Default Author:    [ Mark LaCasse, randvax!lacasse ].
   Default Date:      [ 84/01/05 ].

   Define Enumeration Type-lookahead-opponent: BR1, BR2.
           [ BR1 is Blue's Red version one ]
           [ BR2 is Blue's Red version two ]
   Declare Lookahead-opponent by example:
       Let Lookahead-opponent be Type-lookahead-opponent.
         Definition: [ Blue's Red, opposes Blue in Lookaheads ].

   Declare Time-to-wake by example:
       Let Time-to-wake be 45786.
         Prompt String:
          "Enter the date and time in the format: MMM DD, hh.mm".
         Validation Function: Check-time-input.
         Definition: [ General purpose, future time to wake Blue ].

End Declarations.
```

X. COPROCESSES

A coprocess is an executing program (a process) that is started by another executing program (its parent). The two then execute independently and asynchronously of one another This section discusses how coprocesses are created, put to sleep, and terminated.

CREATING A COPROCESS

Coprocesses are created by calling a built-in RAND-ABEL function called Startup-plan. It takes two arguments, *plan-start* and *plan-name*. *Plan-start* is given an object representing the top-level function of the to-be-created process. This object can be created by use of the function expression. *Plan-name* is a string identifying the new process. The function Startup-plan returns an object that is of type process. Therefore, the initiation of a new process might be performed by a statement such as the following:

```
Declare new-proc:  Let new-proc be Self.

Let new-proc be Report from Startup-plan
                 using (Function Top-of-plan) as plan-start
                  and "Offensive strategy" as plan-name.
```

Note the use of the expression (Function Top-of-plan) to create an object representing the user's function named "Top-of-plan", which is to be executed as the beginning of the new process.

The execution of the Startup-plan function is the only method by which a value of type process can be created.

PUTTING A PROCESS TO SLEEP

A process can cause itself to "go to sleep"--that is, to stop processing until it is awakened by some external program. This is done by calling a built-in function called "Sleep". Sleep takes no arguments and returns no value:

```
Perform Sleep.
```

There are no facilities within the RAND-ABEL language itself for awakening a function once it is sleeping; at present, those facilities are part of the support environment in which a RAND-ABEL process resides and must be invoked directly within that support environment. (See Appendix A for some further information on the support environment for RAND-ABEL.)

TERMINATING A COPROCESS

To terminate a coprocess, the "Remove-plan" function is used. It takes one argument, called process: an object of type process that identifies the process to be terminated. Therefore, to terminate the process created by the example above, one would write:

```
Perform Remove-plan using new-proc as process.
```

RESERVED COPROCESS VARIABLES: SELF AND PARENT

The RAND-ABEL system contains two reserved process-type variables: Self and Parent. Self is always equal to the current process. Parent always refers to the parent process that spawned a given process. These variables can also be used in Declare statements as examples of processes, so that new variables of type process can be declared.

RULES FOR THE USE OF COPROCESSES

The following rules govern the use of coprocesses.

1. A coprocess may not call Remove-plan on itself.

2. Processes can be the object of assignment statements, so that statements such as:

   ```
   Let me be Self.
   ```

   are valid.

3. Processes can be values of suitably defined variables or arrays. This was illustrated in above example, where "me" is a variable taking on a process as its value.

4. Processes can be arguments of suitably declared arrays. For example:

```
Declare active_process:
    Let active_process be Self.

Declare array:  Let array of active_process be 17.
```

5. Processes can be parameters of functions. For example, the built-in function "Remove-plan" has as its single parameter an object of type process.

6. Processes are subject to no other operations.

In the RSAS environment, coprocesses correspond to decisionmaking agents or other simulation models. At the top level, a system monitor controls the execution of coprocesses.

A given agent (coprocess) may create several subordinate coprocesses to create an organizational hierarchy (such as a hierarchy of command in a military command structure as demonstrated by RSAS decision models).

XI. TOP-LEVEL RAND-ABEL DECLARATIONS, DEFINITIONS, AND STATEMENTS

The RAND-ABEL "language" is a complete programming language. However, only certain of the RAND-ABEL statements and declarations can occur at the top level of a RAND-ABEL program. All other RAND-ABEL constructions occur within these top-level statements and declarations.

The only RAND-ABEL declarations and statements that can occur at the top level are:

- Any valid RAND-ABEL Declare
- Any valid RAND-ABEL Define
- The RAND-ABEL *statements*:

```
Trace If.
Trace Function.

Untrace If.
Untrace Function.
```

- One of the two mutually exclusive statements:

```
Owner: name.
Owner: Global.
```

- A special top-level-only Declare:

```
Declare Ignore name . . . name.
Declare Ignore name, . . . , name.
```

- The set of declarations providing information to the Data Dictionary facility (see Sec. IX).

```
Begin Declarations.
    declaration
       DDdeclaration . . . DDdeclaration
    declaration
       DDdeclaration . . . DDdeclaration
    . . .
End Declarations.
```

The meanings of all normal RAND-ABEL Declares, Defines, and *statements* are found in earlier sections of this document. The meaning of the ownership statements (Owner) is found in Section IX; when used here, the statements declare the ownership tag to be put on all subsequent RAND-ABEL code.

The Declare Ignore statement is used to add a set of *identifiers* to a list (initially null) that the RAND-ABEL Translator will ignore whenever they are encountered. These identifiers can then be used as "noise words" in RAND-ABEL statements, presumably to increase their readability.

For example, the declaration:

```
                Declare Ignore a, an, the.
```

allows one to write a RAND-ABEL statement such as:

```
        Let the color of a piece be white.
```

which is equivalent to the RAND-ABEL statement using explicit comments:

```
        Let [the] color of [a] piece be white.
```

or, the more brutally simple:

```
        Let color of piece be white.
```

Care should be taken in the declaration and use of such noise words, since readers of RAND-ABEL code might overlook the Declare Ignore statement and believe that these words are part of valid RAND-ABEL syntax, possibly leading them to write incorrect RAND-ABEL programs. Also, it should be noted that in normal English the phrases "a piece" and "the piece" mean quite different things, whereas they do not in a RAND-ABEL program in which both "a" and "the" are declared to be noise words, again creating the possibility of confusion in the RAND-ABEL reader's mind.

Appendix A

LOCAL SUPPORT ENVIRONMENT FOR RAND-ABEL

This appendix briefly discusses the use of RAND-ABEL at The RAND Corporation. Its contents are specific to this site. Currently, RAND-ABEL is used at RAND only in the context of the RAND Strategy Assessment System (RSAS). Consequently, the following guidelines are RSAS-specific.

The RAND-ABEL Translator is generally used in two modes: (1) as an aid in preparing syntactically correct RAND-ABEL rules, and (2) to produce compilable C code for actual incorporation into the executable RSAC model. The basic difference between (1) and (2) is the handling of the Data Dictionary.

Essentially, in preparing RAND-ABEL programs data items are frequently added, changed, or removed; as a result, the writer cannot use the master data dictionary, but must use his/her own extract of it. This seeming inconvenience can actually be an aid, as it forces the writer to be aware of how his/her program integrates with the other parts of the model.

In all cases the writer incorporates the Data Dictionary into the program by the RAND-ABEL **Include** statement (described in Sec. VIII). Since this mechanism allows nesting (that is, included files can contain other include statements), a two-level approach is used. At the top of a RAND-ABEL program file is an **Include** for a single Data Dictionary file; this file in turn contains **Includes** for all needed Data Dictionary components. In the finished program, the program file instead incorporates the master Data Dictionary file.

All Data Dictionary files end in ".D", while all RAND-ABEL program files end in ".A". The RAND-ABEL Translator is applied to the ".A" files only (with the **Includes** introducing the Data Dictionary). The result of applying the RAND-ABEL Translator is (possibly) a series of error messages and two files: a ".A.c" file and a ".A.I" file. These files are then used to build the integrated RSAS model.

All Data Dictionary and RAND-ABEL program files are eventually registered with the RSAC Data Dictionary administrator for the final integration.

Running the RAND-ABEL Translator itself is quite simple. Applying it to the file "rules.A" would involve typing (on a Sun workstation[1] used by RSAS):

```
/pl/install/bin/enabel rules.A
```

Most people will probably want to use an alias for this and so will place the line:

```
alias enabel /pl/install/bin/enabel
```

into their ".cshrc" files.

In reviewing the error messages produced by the RAND-ABEL Translator, two things should be kept in mind:

(1) The range of legal RAND-ABEL-language inputs is quite large; sometimes it takes several keywords for the enAbeler to detect that an error has occurred. Thus, not only should the line number and word printed by the RAND-ABEL Translator be checked for the error, but the code immediately preceding it as well.

(2) A problem with block-structured languages (like RAND-ABEL) in general, and with top-down parsers like the RAND-ABEL Translator in particular, is a difficulty in recovering from certain errors. (That is, there is often a problem with finding the start of legal statements after the error and in recovering context skipped over because of the error.) As a result, it is quite possible for a single error to produce a hundred error messages. Thus, there are occasions where only the first error reported is an actual problem, while subsequent error messages are a result of declarations, definitions, or statement boundaries having been missed. Note that this is not always the case. If a several-line gap appears between errors, there is a good chance that the later error is valid. However, an

[1]Sun workstations running Sun operating system Release 3.2 (a System V/Berkeley UNIX derivative operating system) are supported.

error in a declaration can cause spurious errors wherever the item being declared is subsequently used, even if thousands of error-free lines intervene.

In all cases, the RAND-ABEL program writer has to do a certain amount of learning through experience and through sharing experience with other writers. Also, there are certainly cases where the RAND-ABEL Translator can do a better job of finding and reporting errors, and in recovering from them. Communication is thus very important.

CHANGING RAND-ABEL RULES (SOURCE CODE)

To make changes to the rules governing program execution, the analyst must edit the relevant RAND-ABEL source file (denoted by a ".A" suffix on the filename). Those changes can be checked for correct syntax and incorporated into the RSAS game by either of two methods. The first involves the use of the RAND-ABEL Interpreter, while the other compiles the code using the so-called "Enabeler" or RAND-ABEL Translator. These methods are described briefly below.

Interpreting Source Code

The fastest way to incorporate new rules into the RSAS gaming environment is to move the functions containing those rules into a special Interpreter directory called INT under Rsas/Run. Anytime a ".A" RAND-ABEL source file is created or modified in that directory, the RSAS monitor will use that code, instead of the compiled version of the source code, when the game is star' ·d or restarted from the RSAS Control Panel.

Some words of caution will help to avoid confusion and mistakes when moving RAND-ABEL rules to the INT directory. First, copy the source files containing the functions of interest to the INT directory. Then, edit the interpreted files to remove (1) all Include "filename" statements in the file, and (2) any other functions that have been left unchanged from their compiled counterparts. Removing unchanged functions simply reduces the amount of code interpreted and results in faster execution, since compiled code executes approximately an order of magnitude faster than interpreted code. Also, make certain that there is an Owner: statement at the beginning of the file (Owner: Blue." for example).

The file ".defines" contains all #Define macro statements identical to the ones in Rsas/Init/incl.D that are used during compilation. This file, along with the Data Dictionary itself, is automatically included by the Interpreter.

To temporarily prevent interpretive executic· of a source file, move the source file to the Hide directory that exists under the INT directory. When it is desired to interpret the source file again, merely move it back up to the INT directory.

The following diagram illustrates the relationship of source directories to the interpreted source directory.

Source and Interpreter Directory Structure

```
                    Rsas
                    /  \
                 Src    Run
                 /        \
      NCL  Awp  Green ...    INT
      / \                      \
   Red   Blue                   Hide
```

When the System Monitor begins or resumes running a game, it will first check the INT directory for source files to interpret and invoke the Interpreter to process these files. If the files are free of syntax or other errors, the Interpreter will log a message in the scrollable text window under the Control Panel (the game log window) indicating that the functions in these files will be interpreted, and System Monitor will then continue the game. Otherwise, error messages will appear in the game log window and the game will stop immediately. If the game is resumed a second time, no further files will be interpreted (until the user edits them correcting the errors), and the game will continue using the compiled versions of the functions.

Compiling and Loading Source Code

There are two steps involved in producing an executable RSAS from the RAND-ABEL source files. The first step is to "enabel" the source code, checking syntax, producing C code and eventually ".o" object files. The next step is to combine the various ".o" modules into a single executable (binary) file called run.sun in Rsas/bin directory.

Incremental Changes to Source Code

Changes to a source file can be incorporated into RSAS runs by compiling it and loading the resulting output with the existing compiled modules. See the instructions above for determining which configuration file to touch[2] in order to compile a set of .A files.

In order to build a new executable RSAS file, use the makefile in the top-level Rsas directory. By typing "make" in that directory, all source code will be checked for modifications (the updated files will be recompiled) and loaded into a new "run.new" executable file in the Rsas/bin directory. Renaming that file to "run.sun" will cause it to be used in subsequent RSAS runs.

Full Data Dictionary Remake

When changes are made to Data Dictionary declarations (e.g., by adding a new variable or enumeration type), it is necessary to remake all RAND-ABEL source files. This can be done by "touching" the dictionary.D file in Rsas/Init directory and starting the RSAS "make" from the top-level Rsas directory as shown below (assuming the current working directory is Rsas):

```
cd Init
touch dictionary.D
cd ..
make
```

[2] Touch is a UNIX command.

A "full" Data Dictionary remake will also produce a new World Situation Data Set (WSDS) file called "wsds.new" in Rsas/Run/Wsds directory that should be subsequently renamed to "wsds" to use it in an RSAS run. In addition, a "scdb.new" file is produced in Rsas/Run that should be renamed "scdb.S" for using the Source Code Data Base (also known as the Cross Reference Tool).

Appendix B

QUICK REFERENCE GUIDE TO THE RAND-ABEL LANGUAGE

KEYWORDS

The following is a list of RAND-ABEL keywords. Words that always occur in sequence as phrases are shown together; words are shown separately that are optional or are one of several possible choices within a phrase.

and
are
are not
as
Attribute
Author

Begin Declarations
Break
by

Clone
Comments
Concatenated with
Constant
Continue

Date
Declare
Declare...by example
Decrease by
Default
Define
#define
Definition
Divide...by
Divided by

End
End Declarations
Erasable
Erase
Evaluate
Everyone
Exit

For
Format
from
Function

Global

If...Then
If...Then...Else
Ignore
in
Include
Increase...by
Initialize
is
is at least
is at most
is greater than
is less than
is not

Let...be
Log

Macro
Make
Method
minus
modulo
Multiply...by

negative
No
Not

of
or
Owner

Parent
Perform
plus
Pointer to
Print
Prompt

Range
Read
Record
References
Report from
Reporting

Self
Semi-erasable
Status
String

Table
There is
times
Trace

Unerasable
Unspecified
Untrace
Use
using

Validation
Value of

While
with
Write

Yes

In addition, the following special symbols also act as keywords having special meaning:

@	--	(...)	{...}	[...]	,	:	.
+	-	*	/	%	~	$	
=	~=	>=	<=	>	<	&	|

In the above lists, ellipses (...) are used to represent intervening words in a standard phrase. (Note: Elsewhere in this manual, ellipses are used to represent certain syntactic options; see item #4, below.)

BNF DESCRIPTION OF RAND-ABEL

To describe the RAND-ABEL language in a concise format for easier scanning, the Backus-Naur Form (BNF) of RAND-ABEL is shown below. The following notational conventions are used in this BNF. Nonterminals in the language are denoted by names with a capitalized first letter (e.g., Abel), while terminal symbols are all uppercase (e.g., ENUMERATION). Note that the nonterminals that represent keywords are shown here in all uppercase, whereas in a RAND-ABEL program only the first character may be capitalized. A few special symbols, namely

```
  ::=    |    anyname*    <empty>
```

are part of the notation, NOT part of the RAND-ABEL programming language. Star ''*'' immediately follows any quantity, represented by ''anyname'' above, that can be repeated zero or more times. The ''|'' separates alternative selections that are valid in the same construct. <empty> indicates that an option may be omitted entirely.

```
Abel            ::= Component*
Component       ::= Declaration
                 |   Definition
                 |   OWNER : NAME
                 |   OWNER : GLOBAL
                 |   Trace_request IF Opt_file_clause .
                 |   Trace_request FUNCTION Opt_file_clause .
                 |   Trace_request NAME TABLE Opt_file_clause .
                 |   Trace_request LET Opt_file_clause .
                 |   TRACE TO FILE STRING .
                 |   Dictionary
Declaration     ::= DECLARE NAME : Assignment .
                 |   DECLARE NAME : Call .
                 |   DEFINE ENUMERATION : Member_list .
                 |  DEFINE ENUMERATION NAME : Member_list .
Definition       ::= DEFINE NAME Opt_use_clause : Opt _declarations Statements
END
Opt_use_clause   ::= <empty> | Paramexpr
Opt_declarations::= Declaration*
Statements       ::= Statement*
Statement        ::= .
                 |  Block
                 |  Assignment .
                 |  INCREASE Expr BY Expr .
                 |  DECREASE Expr BY Expr .
                 |  MULTIPLY Expr BY Expr .
                 |  DIVIDE Expr BY Expr .
                 |  Call .
                 |  Io_call .
                 |  Conditional Statement
                 |  Conditional Statement ELSE Statement
                 |  WHILE Expr : Statement
                 |  FOR NAME : Statement
                 |  FOR NAME Expr : Statement
                 |  FOR NAME PREPOSITION Expr : Statement
                 |  FOR NAME NOT PREPOSITION Expr : Statement
                 |  BREAK .
                 |  CONTINUE .
                 |  EXIT .
                 |  EXIT REPORTING Expr .
                 |  Trace_request IF Opt_file_clause .
                 |  Trace_request FUNCTION Opt_file_clause .
                 |  Trace_request NAME TABLE Opt_file_clause .
                 |  Trace_request LET Opt_file_clause .
                 |  TABLE Element . Opt_newlines Rows .
                 |  TABLE Block . Opt_newlines Rows .
                 |  NAME TABLE . Opt_newlines Rows .
Block            ::= { Opt_declarations Statements }
Assignment       ::= LET Expr BE Expr
```

```
Call            ::= PERFORM Element Opt_use_clause
Io_call         ::= Io_primitive Expr_list
                |   Io_primitive WITH Expr Expr_list
                |   Io_primitive NAME WITH Expr Expr_list
Io_primitive    ::= PRINT
                |   LOG
                |   EXPLAIN
Conditional     ::= IF Expr THEN
Trace_request   ::= TRACE
                |   UNTRACE
Opt_file_clause ::= <empty> | TO FFILE STRING
Expr_list       ::= Expr*
Row             ::= Expr_list NEWLINE Opt_newlines
Rows            ::= Row*
Expr            ::= Logexpr
                |   Eval_clause Expr_list
Eval_clause     ::= EVALUATE
                |   EVALUATE WITH Factor
Logexpr         ::= Logterm
                |   Logexpr OR Logterm
Logterm         ::= Logfactor
                |   Logterm AND Logfactor
Logfactor       ::= Subexpr
                |   Logfactor EQUAL Subexpr
                |   Logfactor NOT_EQUAL Subexpr
                |   Logfactor LESS_THAN Subexpr
                |   Logfactor GREATER_THAN Subexpr
                |   Logfactor LESS_OR_EQUAL Subexpr
                |   Logfactor GREATER_OR_EQUAL Subexpr
Subexpr         ::= Term
                |   Subexpr + Term
                |   Subexpr - Term
                |   Subexpr DOLLAR Term
Term            ::= Factor
```

```
                  |  Term * Factor
                  |  Term / Factor
                  |  Term MODULO Factor
Factor           ::= Element
                  |  Element IS IN Element
                  |  REPORT FROM Element
                  |  REPORT FROM Element Paramexpr
                  |  Element PREPOSITION Domlist
                  |  -Element
                  |  NOT Element
                  |  =Element
                  |  ~=Element
                  |  <Element
                  |  >Element
                  |  <=Element
                  |  >=Element
                  |  MAX Element
                  |  MIN Element
Domlist          ::= Factor
                  |  Factor COMMA AND Domlist
                  |  Factor COMMA Domlist
Paramexpr        ::= USING Expr As_for NAME Paramlist
Paramlist        ::= <empty>
                  |  COMMA Expr As_for NAME Paramlist
                  |  COMMA AND Expr As_for NAME Paramlist
As_for           ::= AS
                  |  FOR
Element          ::= VALUE OF Element
                  |  POINTER TO Element
                  |  FUNCTION Element
                  |  ( Expr )
                  |  { NAME SUCH THAT Expr }
                  |  { }
                  |  { Member_list }
```

```
                    |   STRING
                    |   FLOAT
                    |   INTEGER
                    |   BOOLEAN
                    |   UNSPECIFIED
                    |   UNIVERSE
                    |   NAME
                    |   Poss_clause NAME
Member_list         ::= NAME
                    |   NAME Member_list
                    |   NAME COMMA Member_list
Poss_clause         ::= POSSESSIVE
                    |   Poss_clause POSSESSIVE
Opt_newlines        ::= NEWLINE*
Dictionary          ::= BEGIN DECLARATIONS . Dict_entries END DECLARATIONS .
Dict_entries        ::= Dict_entry
                    |   Dict_entries Dict_entry
Dict_entry          ::= DEFAULT Descriptor
                    |   Declaration Description
                    |   Nodefault_decl
Description         ::= Description*
Descriptor          ::= Ownership
                    |   Access_method
                    |   Macro_method
                    |   Func_method
                    |   Usage
                    |   Init_struct
                    |   Access_type
                    |   Prompt_func
                    |   Prompt_string
                    |   Val_func
                    |   Val_range
                    |   Format
                    |   Author
```

```
                |   Date
                |   Informative_def
                |   References
                |   Comments
                |   Status
Ownership       ::= OWNER : NAME .
                |   OWNER : Poss_clause NAME .
                |   OWNER : GLOBAL .
                |   OWNER : EVERYONE .
Access_method   ::= METHOD : DIRECT .   .
                |   METHOD : MACRO .
                |   METHOD : FUNCTION .
Macro_method    ::= MACRO : STRING .
Func_method     ::= FUNCTION : NAME .
                |   FUNCTION : Poss_clause NAME .
Usage           ::= USE : Use_type .
Use_type        ::= CLONE
                |   NOCLONE
                |   CONSTANT
Init_struct     ::= INITIALIZE .
                |   NO INITIALIZE .
Access_type     ::= READ : Group_list .
                |   NOREAD : Group_list .
                |   WRITE : Group_list .
                |   NOWRITE : Group_list .
Group_list      ::= EVERYONE
                |   Group
                |   Group_list COMMA Group
Group           ::= NAME
                |   Poss_clause NAME
Prompt_func     ::= PROMPT FUNCTION : NAME .
                |   PROMPT FUNCTION : Poss_clause NAME .
Prompt_string   ::= PROMPT STRING : STRING .
Val_func        ::= VALIDATION FUNCTION : NAME .
```

```
                 |   VALIDATION FUNCTION : Poss_clause NAME .
Val_range        ::= VALIDATION RANGE : Number TO Number .
Number           ::= INTEGER
                 |   FLOAT
Format           ::= READ FORMAT : STRING .
                 |   WRITE FORMAT : STRING .
Author           ::= AUTHOR : [ STRING ] .
Date             ::= DATE : [ STRING ] .
Informative_def  ::= DEFINITION : [ STRING ] .
References       ::= REFERENCES : [ STRING ] .
Comments         ::= COMMENTS : [ STRING ] .
Status           ::= STATUS : [ STRING ] .
Nodefault_decl   ::= NO DEFAULT METHOD .
                 |   NO DEFAULT USE .
                 |   NO DEFAULT INITIALIZE .
                 |   NO DEFAULT READ .
                 |   NO DEFAULT WRITE .
                 |   NO DEFAULT PROMPT FUNCTION .
                 |   NO DEFAULT PROMPT STRING .
                 |   NO DEFAULT VALIDATION FUNCTION .
                 |   NO DEFAULT VALIDATION RANGE .
                 |   NO DEFAULT OWNER .
                 |   NO DEFAULT READ FORMAT .
                 |   NO DEFAULT WRITE FORMAT .
                 |   NO DEFAULT AUTHOR .
                 |   NO DEFAULT DATE .
                 |   NO DEFAULT DEFINITION .
                 |   NO DEFAULT REFERENCES .
                 |   NO DEFAULT COMMENTS .
                 |   NO DEFAULT STATUS .

Comment          ::= [ STRING ]
```

SUMMARY OF RAND-ABEL SYNTAX CATEGORIES

The following pages contain a summary of the RAND-ABEL syntax charts contained within this document.

```
+-expression-----------------------------------------------------+
|                                                                |
|  Report from function-invocation                               |
|                                                                |
|  Evaluate unitparam  . . .  unitparam                          |
|  Evaluate with format-spec unitparam  . . .  unitparam         |
|                                                                |
|  unary-operator expression                                     |
|  expression binary-operator expression                         |
|                                                                |
|  simple-expr                                                   |
|                                                                |
+----------------------------------------------------------------+
+-simple-expr-------------------------------------------------+
|                                                             |
|  Function function-name                                     |
|                                                             |
|   +-unitparam-------------------------------------------+   |
|   |                                                     |   |
|   |  Yes                                                |   |
|   |  No                                                 |   |
|   |  enumerated-value                                   |   |
|   |  numeric-literal                                    |   |
|   |  quoted-string                                      |   |
|   |  variable-name                                      |   |
|   |  Unspecified                                        |   |
|   |        --                                           |   |
|   |        lv                                           |   |
|   |  ( expression )                                     |   |
|   |                                                     |   |
|   +-----------------------------------------------------+   |
|   +-lvalue----------------------------------------------+   |
|   |                                                     |   |
|   |   variable-name                                     |   |
|   |                                                     |   |
|   |   Occupant of variable-name                         |   |
|   |   Occupant of array-access                          |   |
|   |                                                     |   |
|   |   Pointer to variable-name                          |   |
|   |   Pointer to Attribute array-name                   |   |
|   |                                                     |   |
|   |   +-array-access--------------------------------+   |   |
|   |   |                                             |   |   |
|   |   |  array-name of simple-expr ,      . . .     |   |   |
|   |   |             in             , and            |   |   |
|   |   |             by             , and            |   |   |
|   |   |                                             |   |   |
|   |   |                      . . .  ,   simple-expr |   |   |
|   |   |                             , and           |   |   |
|   |   |                                             |   |   |
|   |   +---------------------------------------------+   |   |
|   +-----------------------------------------------------+   |
+-------------------------------------------------------------+
+-declaration---------------------------------------------------------+
|                                                                      |
|    Declare variable-name:                                            |
|    Declare variable-name by example:                                 |
|                                                                      |
|        Let variable-name be expression.                              |
|        Let variable-name be identifier constant.                     |
|        Let variable-name be enumerated variable                      |
|                                                                      |
|    Declare array-name:                                               |
|    Declare array-name by example:                                    |
|                                                                      |
|       Let array-name of simple-expr ,        . . .                   |
|       Let array-name in simple-expr , and  . . .                     |
|       Let array-name by simple-expr , and  . . .                     |
|                                                                      |
|                                    . . .    , simple-expr            |
|                                    . . .    , and simple-expr        |
|                                                                      |
|                              be expression.                          |
|                                                                      |
|   Declare func-name :                                                |
|   Declare func-name by example:                                      |
|                                                                      |
|        Let expression be Report from named-function-call.            |
|        Perform named-function-call.                                  |
|                                                                      |
+----------------------------------------------------------------------+
+-function-definition---------------------------------------------+
|                                                                  |
| Define named-function-call : declaration . . .                  |
|                              declaration                        |
|                                statement . . .                  |
|                                statement                        |
|                                                                  |
| End.                                                             |
|                                                                  |
+------------------------------------------------------------------+

+-named-function-call---------------------------------------------+
|                                                                  |
|  func-name                                                       |
|                                                                  |
|  func-name  using expression as    param-name , . . .            |
|                              for               , and            |
|                                                                  |
|                 . . . ,      expression as  param-name           |
|                       , and              for                     |
|                                                                  |
+------------------------------------------------------------------+

+-function-invocation-------------------------------- ---------+
|                                                                  |
|  named-function-call                                             |
|                                                                  |
|  func-ptr                                                        |
|                                                                  |
|  func-ptr   using expression as    param-name ,       . . .      |
|                              for               , and            |
|                                                                  |
|                 . . . ,      expression as  param-name           |
|                       , and              for                     |
|                                                                  |
+------------------------------------------------------------------+
+-statement------------------------------------------------+
|                                                          |
|  Let lvalue be expression.                               |
|  Let pointer be expression.                              |
|                                                          |
|                                                          |
|  Increase  lvalue by expression.                         |
|  Decrease lvalue by expression.                          |
|  Multiply lvalue by expression.                          |
|  Divide lvalue by expression.                            |
|                                                          |
|  If Boolean-expression Then statement                    |
|                                                          |
|  If Boolean-expression Then statement Else statement     |
|                                                          |
|  For variable : statement                                |
|                                                          |
|  While Boolean-expression : statement                    |
|                                                          |
|  Continue.                                               |
|                                                          |
|  Break.                                                  |
|                                                          |
|  Table func-name                                         |
|  Table compound-statement                                |
|  Decision Table                                          |
|                                                          |
|          table-header.                                   |
|                                                          |
|          table-body.                                     |
|                                                          |
|  Perform function-invocation.                            |
|                                                          |
|  Exit.                                                   |
|  Exit Reporting simple-expr.                             |
|                                                          |
|  Print unitparam  . . .  unitparam.                      |
|  Print with format-spec unitparam  . . .  unitparam.     |
|                                                          |
|  Print streamname unitparam  . . .  unitparam.           |
|  Print streamname with format-spec unitparam . . . unitparam. |
|                                                          |
|  Log unitparam  . . .  unitparam.                        |
|  Log with format-spec unitparam  . . .  unitparam.       |
|                                                          |
|  Log streamname unitparam  . . .  unitparam.             |
|  Log streamname with format-spec unitparam . . . nitparam. |
|                                                          |
|  { declaration . . . declaration                         |
|                                                          |
|       statement . . . statement  }                        |
|                                                              |
|   .                                                          |
|                                                              |
|   ()                                                         |
|                                                              |
|   Trace If.                                                  |
|   Trace Function.                                            |
|                                                              |
|   Untrace If.                                                |
|   Untrace Function.                                          |
|                                                              |
+--------------------------------------------------------------+
+-data dictionary specification block-----------------------------+
|                                                                 |
|   Begin Declarations.                                           |
|   [No] Default DDdeclaration ...                                |
|                                                                 |
|      declaration                                                |
|                                                                 |
|            DDdeclaration . . .                                  |
|                                                                 |
|            DDdeclaration                                        |
|                                                                 |
|      declaration                                                |
|                                                                 |
|            DDdeclaration . . .                                  |
|                                                                 |
|            DDdeclaration                                        |
|                                                                 |
|      . . .                                                      |
|                                                                 |
|   End Declarations.                                             |
|                                                                 |
+-----------------------------------------------------------------+

+-DDdeclaration---------------------------------------------------+
|                                                                 |
|   Method: Direct.                                               |
|   Method: Function.                                             |
|   Method: Macro.                                                |
|                                                                 |
|   Function: func-name.                                          |
|                                                                 |
|   Macro: string-literal.                                        |
|                                                                 |
|   Use:    Clone.                                                |
|   Use:  No Clone.                                               |
|   Use:  Constant.                                               |
|                                                                 |
|   Owner:  owner-name.                                           |
|                                                                 |
|   Owner:  Global.                                               |
|                                                                 |
|   Read    Everyone.                                             |
|   Read    owner-name  . . .  owner-name.                        |
|   Read    owner-name , . . . , owner-name.                      |
|                                                                 |
|   Noread  Everyone.                                             |
|   Noread  owner-name  . . .  owner-name.                        |
|   Noread  owner-name , . . . , owner-name.                      |
|                                                                 |
|   Write   Everyone.                                             |
|   Write   owner-name  . . .  owner-name.                        |
```

Write *owner-name* , . . . , *owner-name*.

Nowrite Everyone.
Nowrite *owner-name* . . . *owner-name*.
Nowrite *owner-name* , . . . , *owner-name*.

Read Format: *string-literal*.

Write Format: *string-literal*.

Validation Range: *numeral* to *numeral*.

Validation Function: *func-name*.

Prompt Function: *func-name*.
Prompt String: *string-literal*.

Initialize.

No Initialize.

Author: *comment*.

Date: *comment*.

Definition: *comment*.

References: *comment*.

Comments: *comment*.

Status: *comment*.

No Default Author.
No Default Comments.
No Default Date.
No Default Definition.
No Default Initialize.
No Default Method.
No Default Owner.
No Default Prompt Function.
No Default Prompt String.
No Default Read Format.
No Default Write Format.
No Default References.
No Default Status.
No Default Use.
No Default Validation Function.
No Default Validation Range.

```
+-meta-statement--------------------------------------------+
|                                                           |
|   #Define name [ unquoted-string ].                       |
|                                                           |
|   Include "filename".                                     |
|                                                           |
+-----------------------------------------------------------+
```

INDEX

BIBLIOGRAPHY

Bell Laboratories, *UNIX Programmer's Manual*.

Davis, Paul K., *Applying Artificial Intelligence Techniques to Strategic-Level Gaming and Simulation*, The RAND Corporation, N-2752-RC, June 1988. (Also published in M. Elzas, T. I. Oren, and B. P. Zeigler (eds.), *Modelling and Simulation Methodology in the Artificial Intelligence Era*, Elsevier Science Publishers, B. V. (North-Holland), 1986, pp. 315-338.

Davis, Paul K., Bruce W. Bennett, and William Schwabe, "Analytic War Gaming with the RAND Strategy Assessment System (RSAS)," The RAND Corporation, P-7464, July 1988.

Davis, Paul K., and H. Edward Hall, *Overview of RSAS System Software*, The RAND Corporation, N-2755-NA, forthcoming.

Davis, Paul K., and James A. Winnefeld, *The RAND Strategy Assessment Center: An Overview and Interim Conclusions about Utility and Development Options*, The RAND Corporation, R-2945-DNA, March 1983.

Fain, Jill, D. Gorlin, F. Hays-Roth, S. Rosenschein, H. Sowizral, and D. Waterman, *The ROSIE Language Reference Manual*, The RAND Corporation, N-1647-ARPA, December 1981.

Kernighan, Brian W., and Dennis M. Richie, *The C Programming Language* Prentic-Hall, Englewood Cliffs, New Jersey, 1978.

Schwabe, William, and Lewis M. Jamison, *A Rule-Based Policy-Level Model of Nonsuperpower Behavior in Strategic Conflicts*, The RAND Corporation, R-2962-DNA, December 1982.

Shapiro, Norman Z., H. Edward Hall, Robert H. Anderson, and Mark LaCasse, *The RAND-ABEL Programming Language: History, Rationale, and Design*, The RAND Corporation, R-3274-NA, August 1985.